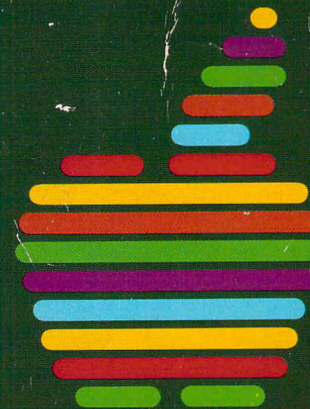
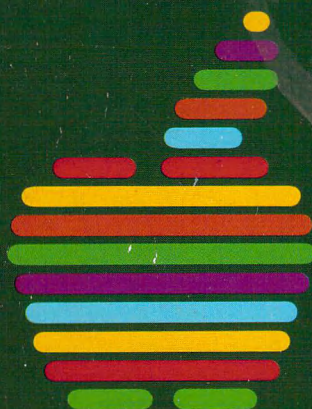


OSBORNE/  
McGRAW-HILL

# APPLE II<sup>®</sup> USER'S GUIDE

BY LON POOLE WITH MARTIN MCNIFF & STEVEN COOK





# **APPLE II USER'S GUIDE**





# **APPLE II USER'S GUIDE**

**BY LON POOLE  
WITH MARTIN McNIFF  
& STEVEN COOK**

**OSBORNE/McGraw-Hill  
Berkeley, California**

Apple, Apple II, Apple II Plus, Disk II, and Applesoft are registered trademarks of Apple Computer Inc. with regard to any computer product.

Published by  
OSBORNE/McGraw-Hill  
630 Bancroft Way  
Berkeley, California 94710  
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write OSBORNE/McGraw-Hill at the above address.

#### **APPLE II USER'S GUIDE**

1234567890 DODO 8987654321  
ISBN 0-931988-46-2

Copyright © 1981 McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publishers.

**PHOTO CREDITS:** All photos by Harvey Schwartz unless otherwise indicated.

**COVER DESIGN:** Timothy Sullivan

## **ACKNOWLEDGEMENTS**

**We gratefully acknowledge:**

**Apple Computer, Inc., for supplying equipment and information needed to write this book, as well as providing a technical review of the manuscript.**

**Mike Lipschutz and the Byte Shop of Hayward, who kindly provided a video monitor on a moment's notice.**

**Robert Thomson, for his research and writing, which provided the foundation for Chapter 8 and the appendices.**

**Janice Enger, for the Blanket program, and the manner in which it is used to illustrate the text.**





# Contents

**Acknowledgments v**

**Introduction xi**

## **1. PRESENTING THE APPLE II 1**

Keyboard and TV 1. Inside the Apple II 2. Memory 4. Cassette Recorder 4. Disk Drive 5. Programs 5. External Device Controllers 6. Game Controls 9. Printer 10. Graphics Tablet 10

## **2. HOW TO OPERATE THE APPLE II 13**

### **Turning the Power On**

What You See on the TV 14. The Prompt Character 15

### **The Keyboard 16**

### **The Cassette Recorder 19**

### **Using the Disk II 21**

The Disk Operating System 22. Preparing Blank Diskettes 27

### **Loading and Running a Program 28**

Use the Right Version of BASIC 28. Loading a Program from Cassette 29. Loading a Program from Disk 29. Starting a Program Running 30. Setting TV Color 30

### **Miscellaneous Components 32**

### **Coping with Errors 32**

Error Messages 32. Correcting Typing Mistakes 32. Accidental Reset 33

## **3. PROGRAMMING IN BASIC 37**

Starting Up BASIC 37

### **Immediate and Programmed Modes 38**

Printing Characters 38. Printing Calculations 39. Error Messages 40. Extra Blank Statements 41. Statements, Lines and Programs 41. Programmed Mode 43. Saving Programs on Cassette 47

**Switching BASICs 49****Advanced Editing Techniques 50**

Deleting Program Lines 50. Adding Program Lines 51. Changing Program Lines 51. Reexecuting in Immediate Mode 56

**Programming Languages 56****Elements of BASIC 57**

Line Numbers Revisited 57. Blank Spaces 58. Variables 62. Arrays 66. Expressions 68

**BASIC Statements 75**

Remarks 76. Assignment Statements 76. Declaring Array and String Size 80. Branch Statements 81. Loops 84. Subroutine 87. Conditional Execution 91. Input and Output Statements 93. Halting and Resuming Program Execution 98

**Functions 99**

Numeric Functions 100. String Functions 102. System Functions 103. User-Defined Functions 103. Function Nesting 104.

**4. ADVANCED BASIC PROGRAMMING 105****Direct Access and Control 105**

Memory and Addressing 105

**Using Peripheral Devices 107****Program Output and Data Entry 108**

More About the PRINT Statement 108. PRINT Formatting Functions 116. Cursor Control and Special Video Effects 119. Text Windows 119. The CHR\$ Function: Programming Characters in ASCII 122. Programming Data Entry 123. Forms Data Entry 136. Formatting Output 142. Programming Printers 148

**Storing Data on Cassette 151****Program Optimization 152**

Faster Programs 152. Compact Programs 153

**Debugging 154****Immediate and Programmed Mode Restrictions 156****5. THE DISK II 157**

About Disks 157. How Data is Stored on Disks 160. Locating Tracks and Sectors 162. Write Protecting 163

**The Disk Operating System 164**

Versions of DOS 164. Initializing Disks 164. Disk Files 164. Diskette Directory 164. Track/Sector List 164. Disk Crash 165

**Booting the Disk II 166**

How to Boot DOS 166

**Beginning Disk Commands 168**

CATALOG 168. LOAD 170. The Disk Version of the RUN Command 170. Specifying the Drive Number 170. Slot Specification 170. Volume Specification 171

**More Disk II Commands 172**

INIT 172. SAVE 174. DELETE 174. LOCK 175. UNLOCK 175. RENAME 175. VERIFY 176

**Using DOS Commands in Programs 176****Using Disk Files 177**

Using Sequential Files 177. How to Append to Sequential Files 186. The POSITION Command 186. Using Random-Access Files 187. A Practical Random-Access Example 187. The Byte Parameter 190

**Other DOS Commands 191**

EXEC 191. MAXFILES 193. Using DOS Debugging Aids 194

**Machine Language (Binary Image) Disk Files 195**

BSAVE 195. BLOAD 196. BRUN 196

## **6. GRAPHICS AND SOUND 197**

### **Low-Resolution Graphics 197**

Setting Up the Graphics Page 198. Graphics Programming Statements 199

### **High-Resolution Graphics 106**

Which Page Should You Use? 203. Setting Up the Graphics Display 204. Alternatives to HGR and HGR2 205. High-Resolution Colors 207. Plotting Points and Lines 208

### **Using High-Resolution Shapes 210**

Defining Shapes 210. Assembling the Shape Table 211. Entering the Shape Table 218. Shape Drawing Commands 221

### **Apple II Sound 224**

Operating the Speaker 225

## **7. MACHINE LANGUAGE MONITOR 231**

Accessing the Monitor 231. Leaving the Monitor 232

### **Functions of the Monitor 233**

Examining the Microprocessor Registers 235. Altering Memory 236. Altering the Microprocessor Registers 238. Saving and Retrieving Memory with Apple II Peripherals 239. Moving and Comparing Blocks of Memory 243. The GO Command 247. Using the Printer 247. The Keyboard Command 248. Setting Display Modes. 248. Eight-Bit Binary Arithmetic Using the Monitor 248. User-Definable Monitor Command 249

### **The Mini-Assembler 250**

Accessing the Mini-Assembler 251. Monitor Commands in the Mini-Assembler 251. Leaving the Mini-Assembler 251. Instruction Formats 251. Using the Mini-Assembler 252. Disassembled Listings 254. Testing and Debugging Programs 255. Integrating Your Program with BASIC 260

## **8. COMPENDIUM OF BASIC STATEMENTS AND FUNCTIONS 261**

Immediate and Programmed Modes 261. BASIC Versions 262. Nomenclature and Format Conventions 262

**Statements (listed alphabetically) 263**

**Functions (listed alphabetically) 312**

## **APPENDICES**

### **A. Derived Numeric Functions 323**

### **B. Editing Commands 325**

### **C. Error Messages 327**

Integer BASIC Error Messages 327. Applesoft Error Messages 328. DOS Error Messages 330

### **D. Intrinsic Subroutines 333**

### **E. Useful PEEK and POKE Locations 339**

### **F. BASIC Reserved Words 345**

Integer BASIC 345. Applesoft 346. DOS 347

### **G. Memory Usage 349**

General Memory Organization 349. The BASIC Language Interpreters 350. DOS Memory Requirements 350. Integer BASIC Memory Usage 350. Applesoft Memory Usage 352

### **H. Disk II Format 355**

The Track/Sector List 356. The Directory 356

### **I. ASCII Character Codes and Applesoft Reserved Word Tokens 359**

### **J. Hexadecimal-Decimal Integer Conversion Table 363**

### **K. Bibliography 371**

### **L. Screen Layout Forms 373**

## **Index 379**





# Introduction

This book is your guide to the Apple II computer. This one book describes the Apple II itself and covers the common peripheral devices and accessories including disk drives and printers.

This book assumes you have access to an Apple II system that is completely hooked up according to the instructions in the appropriate owner's manual provided with each system component. It does not explain how to install your system, but rather how to use it once it is installed.

What *is* an Apple II? How do you make it work? The first two chapters of this book answer those questions. You have probably noticed that the Apple II system is made up of several pieces of equipment all strung together with wires and cables. The first chapter tells you what all the pieces are and what they are good for. The second chapter tells you how to operate each component part. With this knowledge you are ready to use any of the ready-to-run programs that are widely available for word processing, financial analysis, bookkeeping, computer-aided instruction, and entertainment.

The next four chapters of the book teach you how to write your own BASIC programs on the Apple II. Chapter 3 starts things off with a tutorial approach to the fundamentals of both versions of BASIC that are available on the Apple II, Integer BASIC and Applesoft. Chapter 4 continues with coverage of advanced programming topics and BASIC features. Two of the advanced topics, the disk drive and screen display graphics, are important enough to warrant their own chapters. Chapter 5 explains how to use the disk drive to store programs and data

files. Chapter 6 tells you how to program graphics on the display screen using both graphics modes available on the Apple II.

BASIC programs operate on the Apple II under the supervision of the Apple II Monitor. Chapter 7 explains both the standard Monitor and the Autostart Monitor from a BASIC programmer's point of view. The chapter also tells you how to incorporate an assembly language program into your BASIC program.

Chapter 8 contains a complete description of each statement and function available in both versions of BASIC, including disk statements. Along with the appendices, it will serve as a handy reference once you know how to program in BASIC on the Apple II.

# 1

## Presenting the APPLE II

Figure 1-1 is a picture of a typical Apple II computer system. Notice that it takes several separate pieces of equipment to make up a complete system.

Your system may not look exactly like the one pictured in Figure 1-1. Many system components come from a long list of optional equipment. But there are three components that every system has in common: the Apple II itself, the built-in keyboard, and a television. Let's take a closer look at each of these and at some of the more common pieces of optional equipment. We will not describe how to hook up any of these components to the Apple II. For complete installation instructions, refer to the owner's manual supplied with each piece.

### KEYBOARD AND TV

The keyboard and TV screen make communications with the Apple II possible. A standard typewriter-style keyboard comes with the Apple II. It transfers instructions from your fingertips into the Apple II.

The *display screen* can either be an ordinary color television set or a color television monitor. A black-and-white TV works fine too, but of course color displays will show up in black-and-white. The screen not only echoes everything you type so you can visually verify its accuracy, it also displays the reactions of the Apple II to your instructions.

The standard display screen has three different modes of operation. One is for black-and-white text characters only and the other two are chiefly for graphics. In



FIGURE 1-1. A Typical Apple II Computer System

text mode, the standard screen is divided into 24 lines of 40 characters each. The graphics modes deal with points and lines, not characters, and subdivide the screen more finely (graphics are discussed further in Chapter 7).

Most Apple II owners use a television set for their display screen either because they have one or because it's a good excuse to get one. The television monitor produces a sharper picture than a TV set in the computer environment, but you can't use it to watch *Sea Hunt*, *Highway Patrol*, or the 6 o'clock News. Figure 1-2 shows a typical television set hookup.

As you can see, the TV does not connect directly to the Apple II. A slide switch attaches to the TV antenna terminal. With the switch in one position, the TV is its normal erudite self; with the switch in the other position, the TV takes its orders from the Apple II. A cable runs from the slide switch to a special small circuit board which fits conveniently inside the Apple II. It is called an *RF modulator* and it converts the video signal from the Apple II to something your television set can deal with. Your Apple dealer can sell you an RF modulator and show you how to hook it up.

A television monitor requires no RF modulator; it attaches directly to the back of the Apple II as shown in Figure 1-3.

## INSIDE THE APPLE II

The Apple II itself houses the part of the computer that controls the rest of the system — under your guidance, of course! Lurking behind the keyboard are the main Apple II memory banks, the microprocessor, the connection points for all the accessory components, and much more. Figure 1-4 discloses the true identity of these undercover items.

The inside of your Apple II may look a bit different from the one in Figure 1-4.



The basic layout will be the same — the large flat circuit board with dozens of small black *integrated circuits* (also called ICs or chips) in orderly rows and some small circuit boards mounted vertically in the *slots* at the back of the main circuit board. The number of chips and the number and placement of the vertical circuit boards varies from one system to the next.

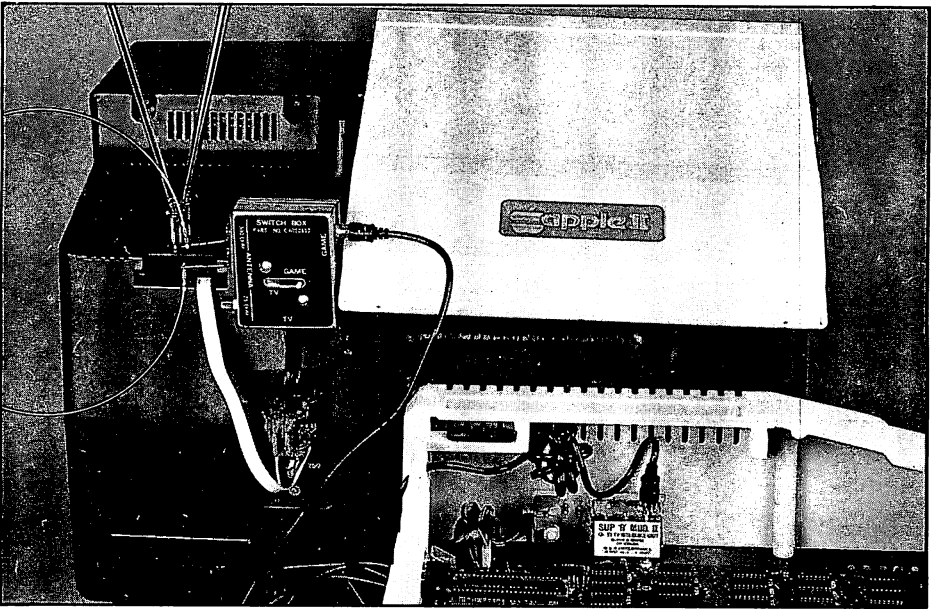


FIGURE 1-2. Television Set Hookup

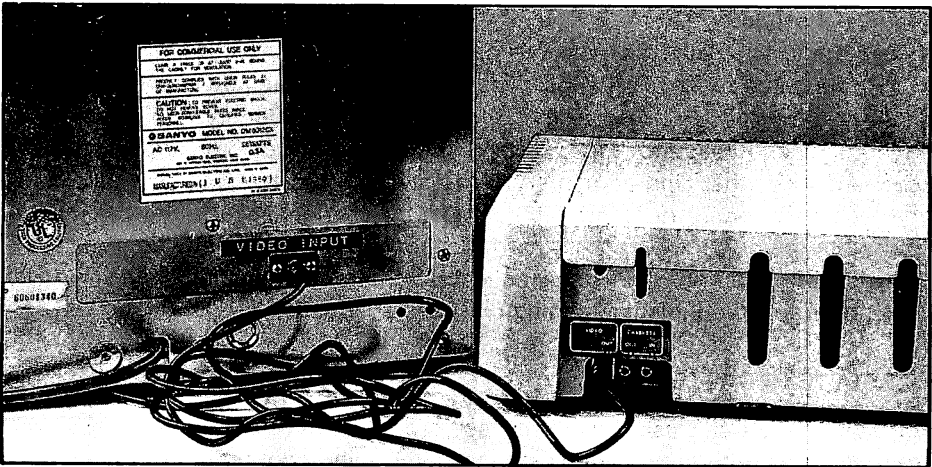


FIGURE 1-3. Television Monitor Hookup

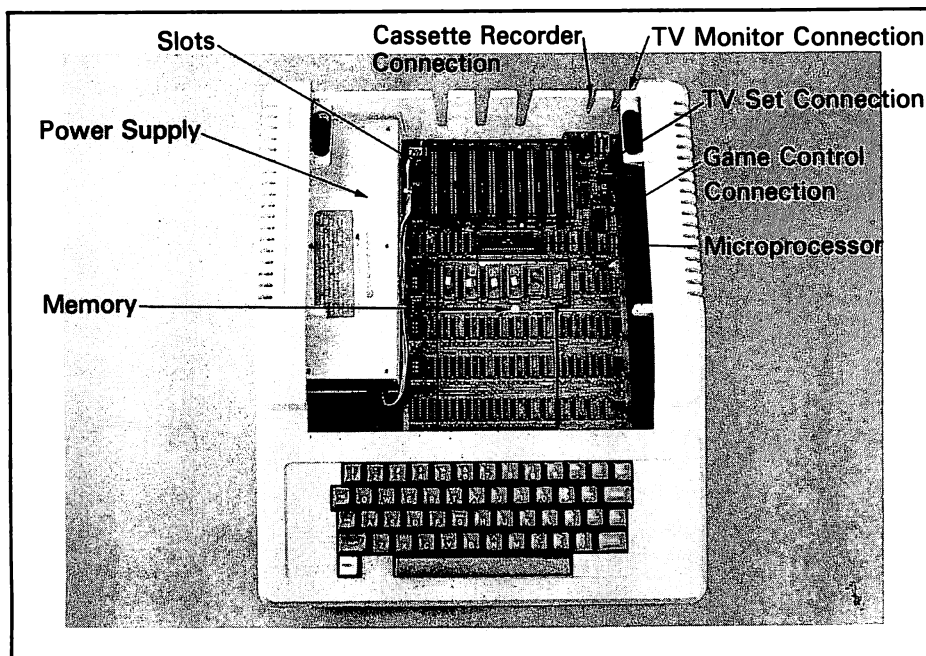


FIGURE 1-4. Inside the Apple II

## MEMORY

Computer memory is typically measured in units called *bytes*. Each byte of memory can hold one character or a similar amount of data. Depending on the number of chips, your Apple II computer has anywhere from 4,096 to 65,536 bytes of memory. This is usually stated 4K to 64K, where K represents 1,024 bytes. The amount of memory available determines how much the Apple II can do, as we will see later.

The Apple II actually has two kinds of memory. One is called *read-only memory* (ROM); its contents never change, even when you turn the power off. ROM contains the programs which give the Apple II its unique identity and enable it to understand and respond appropriately to the commands you type in at the keyboard. The other kind of memory is called *read/write memory* (also called random-access memory or RAM); its contents do change. In fact, the program in read/write memory determines what task the Apple II is currently applied to.

Read/write memory works only as long as the power remains on. As soon as you turn the Apple II off, everything disappears from read/write memory.

## CASSETTE RECORDER

Fortunately, you can use a cassette tape recorder to transfer programs to and from read/write memory, thereby storing a whole library of programs on cassette tape. Figure 1-5 shows a typical cassette recorder installation.

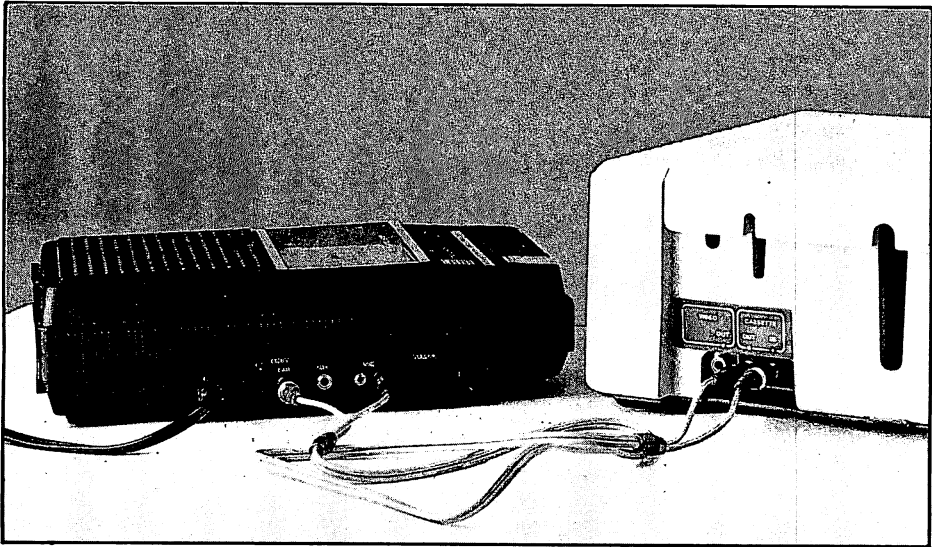


FIGURE 1-5. Cassette Recorder Hookup

## DISK DRIVE

A disk drive far surpasses a cassette recorder as a program storage device. It is more reliable, stores more, and operates faster to boot. What's more, the disk drive easily and quickly stores data such as names and addresses for a mailing list, or correspondence for a word processor. Disk drives come in all shapes and sizes; different models have different storage capacities. Figure 1-6 shows two Disk II units (products of Apple Computer Inc.).

## PROGRAMS

Let's take a slight detour from the guided tour of the Apple II system and look at the different kinds of programs you will use with your system. We're not talking here about the different kinds of things you can do with your Apple II, like word processing, accounting, financial analysis, and so forth, but rather about different classes of programs that must coexist in order for the Apple II to perform any one of these chores. Programs that do things like play games, write letters, and so forth are called *application* programs. They always reside in read/write memory, not ROM. You transfer them to read/write memory from a cassette or disk. So when you want your Apple II to be a word processor, for instance, you use the disk with the word processing application program on it and transfer the program into read/write memory. Chapter 2 explains how to do this.

More often than not, programmers write application programs in a programming language that is easy for them to use but which is too advanced for the Apple II to understand without some help. A special program called an *interpreter*

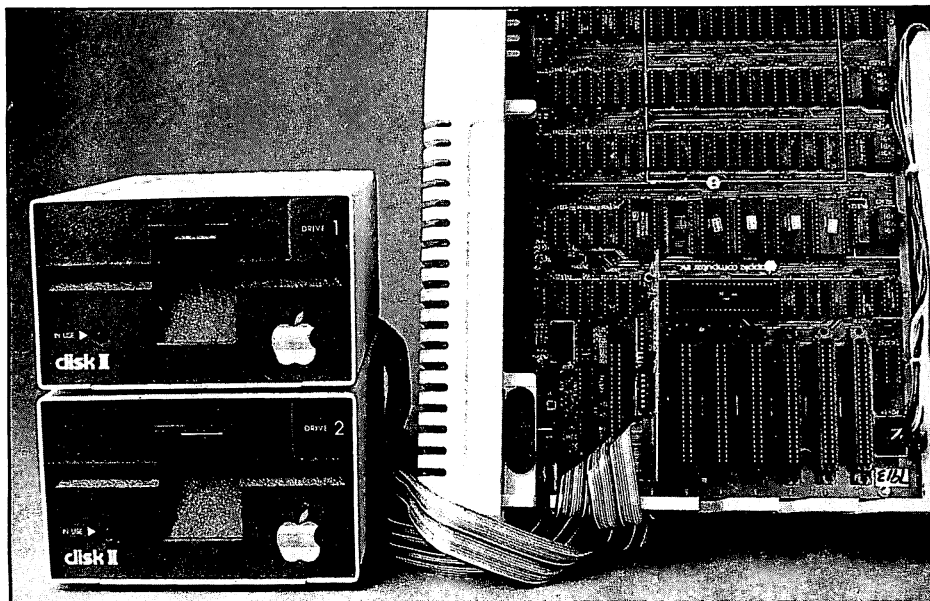


FIGURE 1-6. Disk II Hookup

does just what its name implies. It translates the application program from the language it's written in to a language the computer can understand. The Apple II has a couple of different interpreters, which may exist in either read/write memory or ROM.

The interpreter in turn relies on another program to coordinate the system components. This other program, often called the *operating system* program, performs fundamental system operations like transferring programs from cassette and disk to memory, and echoing keystrokes on the display screen. The Apple II operating system program is known as the *Monitor*. The Monitor always resides in ROM.

## EXTERNAL DEVICE CONTROLLERS

Notice the small vertical circuit boards which plug into the slots at the back of the main Apple II circuit board. The slots exist to accommodate such special circuit boards. The circuit boards, called *controllers* or *cards*, contain extra electronics which enable the Apple II to use peripheral devices such as disk drives.

Apple Computer Inc. makes a number of cards which plug into the slots in the back of the Apple II. Some of them vie for the same slot, but most cards can go into any slot. For the most part, the cards lack scars, birthmarks, or other distinguishing features so it's hard for the uninitiated to tell one from the other. Each card is labeled with its name, but the printing is not always visible when the card is installed. You can usually identify a card from the slot it's in and what's connected



to it. As shown in Figure 1-6, the controller card for the Disk II customarily inhabits slot 6; one or two disk drives attach to it. A second Disk II controller can go in slot 5 for a third and fourth Disk II drive, a third controller card can occupy slot 4 for drives 5 and 6, and so on.

Let's take a look at some of the other cards you may have in your Apple II system.

Each of the three cards shown in Figure 1-7 enhances the programming language capabilities of the Apple II. There are two versions of the programming language BASIC available on the Apple II: Integer BASIC and Applesoft. Installing the appropriate one of these three cards in slot 0 of your Apple II makes it easier to switch from one version of BASIC to another. The *Language System* and *Applesoft Firmware* cards work on any kind of Apple II, but the *Integer BASIC* card is designed only for the Apple II Plus. The Language System card makes other programming languages, like Pascal, available as well.

The *serial interface* card (Figure 1-8) usually inhabits slot 1. Probably the most common device attached to it is a printer.

It's not often that the *parallel printer interface* card (pictured in Figure 1-9) is used concurrently with the serial interface card, because it invariably has a printer attached to it. So the parallel printer interface card usually resides in slot 1.

Figure 1-10 shows the *communications interface* card. With a modem connecting the communications card to the telephone lines, your Apple II can get in touch with other computers at remote sites.

There are more cards available for innumerable tasks from sources other than Apple Computer Inc. For example, you can get cards to control lights and other electrical devices. Another card lets you write programs that electronically synthesize the sounds of musical instruments. There is even a card that works like an extension cord; it uses one slot in the Apple II and provides several slots for additional cards on a separate chassis.

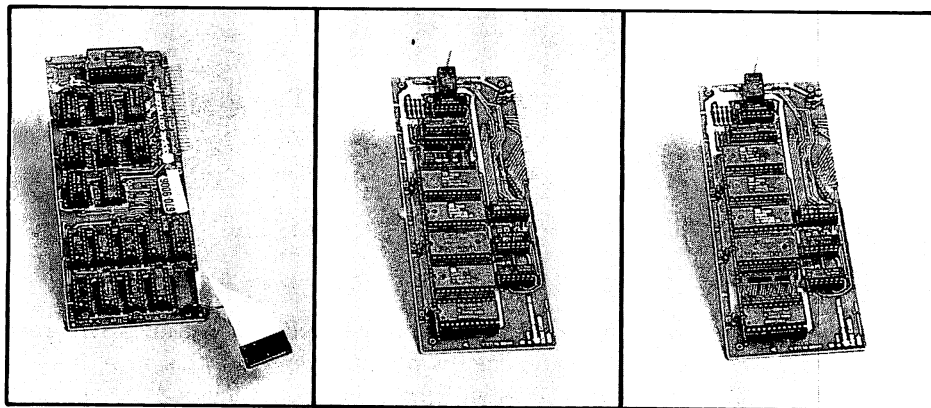


FIGURE 1-7. Left to right: Language System, Applesoft Firmware, and Integer BASIC Cards



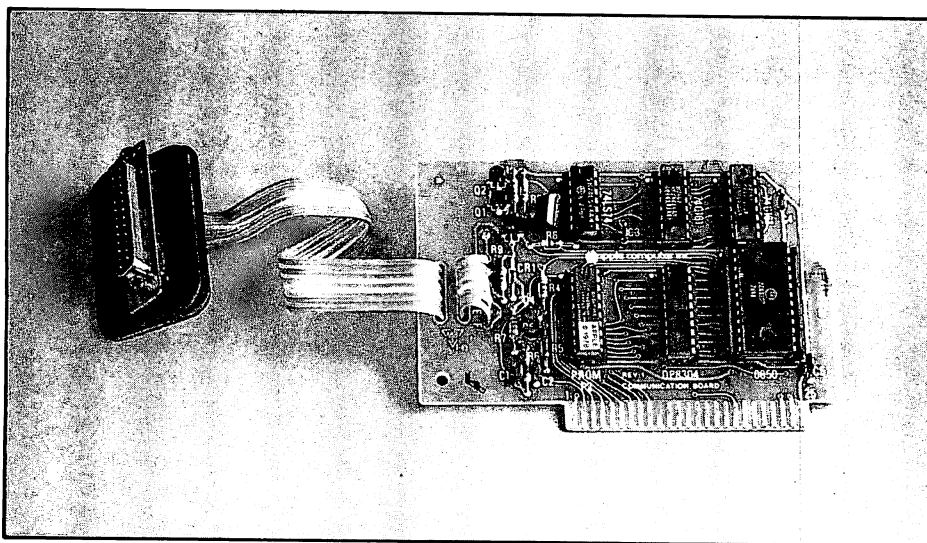


FIGURE 1-10. Communications Interface Card

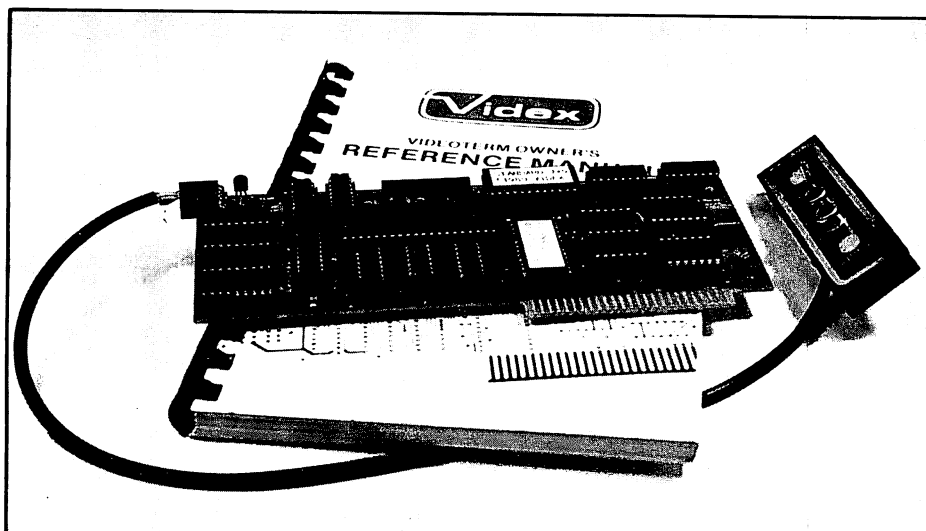


FIGURE 1-11. Special Display Screen Setup

Photo courtesy of Videx.

## GAME CONTROLS

We will conclude the tour of the Apple II innards with a look at the game controls which attach to the Apple II as shown in Figure 1-12. The game controls, or paddles, are used mostly by game programs but may show up in other places as well. Sometimes the game controls are not attached to the Apple II at all.

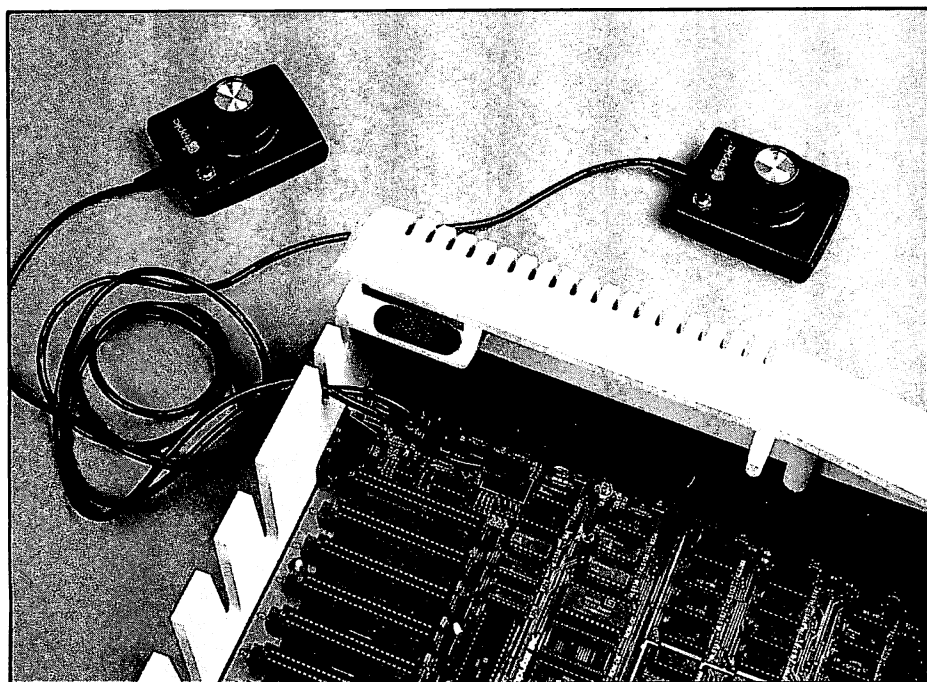


FIGURE 1-12. Game Control Hookup

## PRINTER

Let's turn now to the system printer (see Figure 1-13).

The printer joins the system via either the serial interface card or the parallel printer interface card (depending on what kind of printer it is), usually through slot 1. There are printers of every size, price, and description. Some will print correspondence that looks just as good as anything a typewriter can produce. Others will reproduce your graphics displays (in color, in some cases). Then there are those that are a compromise between the two.

## GRAPHICS TABLET

The graphics tablet from Apple Computer Inc., pictured in Figure 1-14, is a handy device that exploits the graphics capabilities of the Apple II in a special way. With its pen, you can create figures, cartoons, maps, charts, graphs, or any freehand drawing directly on the display screen, in color. Under your direction, it will draw straight lines, solid boxes, and dots. Your drawing can take up the whole screen or just part of it. You can move the drawing around, enlarge it, reduce it, and separate out its individual colors one by one. At any time during this procedure, you can save the screen image on the disk drive for later recall. You can even measure distances with the graphics tablet.

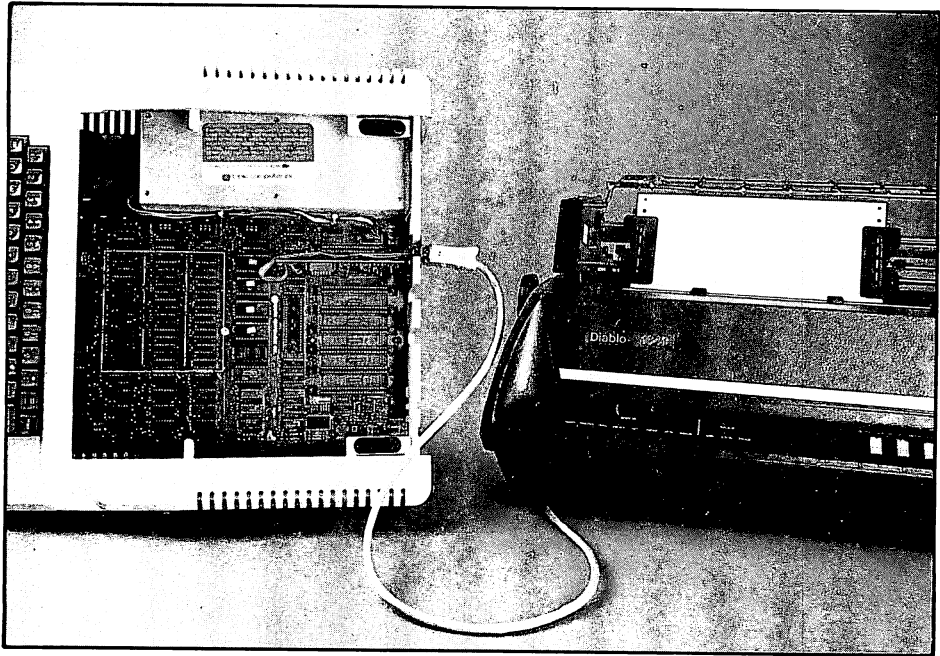


FIGURE 1-13. Printer Hookup

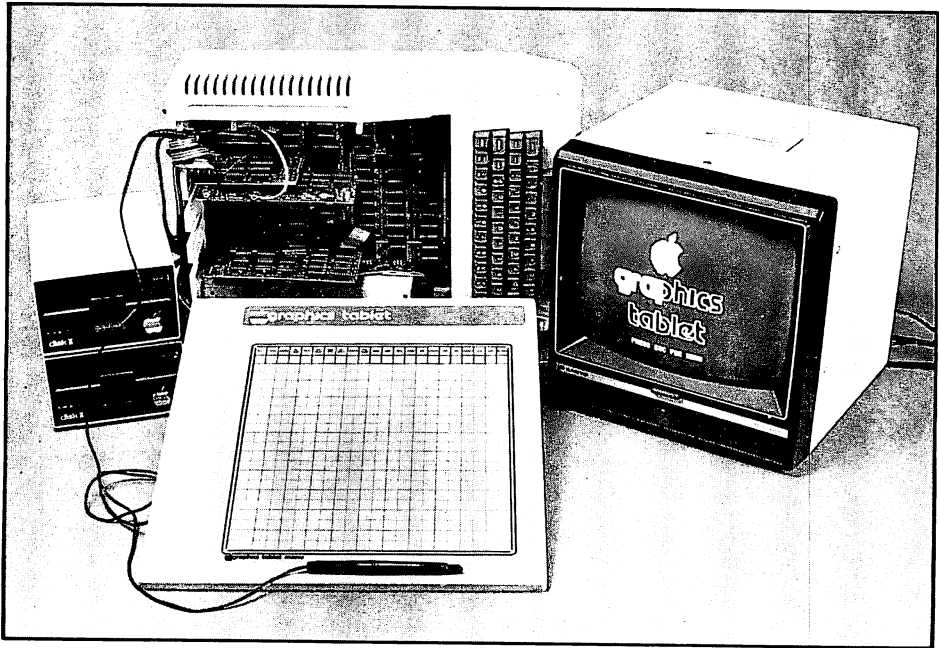


FIGURE 1-14. Graphics Tablet Hookup



# 2

## How to Operate the Apple II

Any computer system can be a bit intimidating when you first sit down in front of it, even if it's all hooked up, as your Apple II system must be before you go any further. This chapter will make you more comfortable around the Apple II by explaining how to use it, but it won't tell you how to set it up. The owner's manuals that come with each piece of equipment have complete instructions in them to help you with the installation procedure. If you need more assistance to be sure you've done it right, check with someone else who uses an Apple II like yours, or with the dealer you bought it from.

### TURNING THE POWER ON

Check that all the system components are connected together correctly and turn on the television set. Turn the volume all the way down. Locate the slide switch hanging from the TV antenna terminals and set it on the GAME or COMPUTER setting. Select the channel specified by the RF modulator instructions (usually channel 33). If you don't know which channel to use, ask someone else who uses the system or a dealer who sells the RF modulator.

Locate the power switch on the rear of the Apple II, next to where the power cord plugs into the computer. Turn the switch ON. You should hear a beep from inside the Apple II. The beep tells you the Apple II is ready. The POWER lamp on the keyboard will be on now unless it's burned out.

If you did not hear a beep, turn the switch OFF, then ON again. If you still do not hear anything, turn the power OFF. Was the POWER lamp lit? If it was not, unplug the Apple and plug in a lamp or a radio to see if the wall outlet has power. Read the instructions again and double check to make sure system components are hooked up properly. If the Apple still won't start, TURN THE POWER OFF! Unplug the Apple and get help from someone with more experience (your dealer). You can do a lot of harm by poking around the insides or switching connecting wires around.

## WHAT YOU SEE ON THE TV

Once the power is on, and you hear the Apple II beep indicating everything is all right, an image will appear on the TV screen.

The exact image will depend on which variety of Apple II you have, but one thing on the screen should be quite obvious because it will be blinking on and off at regular intervals. This flashing white square is called the *cursor*. It marks the location where the next character you type will appear on the screen. The next section explains what to do if you don't see the cursor.

### If You Don't See the Cursor

Your Apple II may have the Language System card installed along with one or more Disk II drives. In this case, you won't see a cursor. Instead there will be some whirring and clacking noises coming from the Disk II drive, and the red IN USE lamp on the front of the Disk II cabinet will light up. For the time being, when this happens press the RESET key. The cursor will appear and the disk drive will quiet down. Go on to the next section.

Some Apple II systems are set up to allow more than 40 characters on each display line. In order to see the cursor on such systems you must type a special command sequence like this:

1. Press the CTRL key and hold it down, press the B key, and then release both keys.
2. For the next command, you need to know which slot the special card (the one the TV monitor attaches to) uses, probably slot 3 or 4. You can open up your Apple II and look inside — slots are numbered 0 through 7 from left to right. Figure 1-11 shows you what to look for. Get help from someone (your dealer) if you're not sure.
3. If your TV monitor attaches to the card in slot 3, type PR#3 and press the RETURN key. If it attaches to slot 4, use PR#4; for slot 2 type PR#2, and so forth.
4. The cursor will now be visible, although it may not be flashing (which is OK). Continue as described below.



TABLE 2-1. Prompt Characters

Prompt Character	Language
*	Monitor
>	Integer BASIC
]	Applesoft

### THE PROMPT CHARACTER

To the left of the cursor there is another character. It is the first character on the line. It could be an asterisk (\*), a greater-than symbol (>), or a right bracket (]). The Apple II is a multilingual computer and the prompt character indicates which language it expects its instructions in. Table 2-1 shows the three prompt characters and their corresponding languages.

#### \* is the Monitor

On many versions of the Apple II the first prompt you will see when you turn the power on is the \* prompt. If this is not the case on your Apple II then you can skip this section and the next.

The asterisk is the prompt for the Apple II *Assembly Language Monitor*. Generally only advanced Apple II users need to communicate directly with the Monitor. If you wish to experiment with the Monitor, you may do so now. While you can't do any harm by playing with the Monitor at this time, you will have to turn the power OFF and then ON again to undo the results of your experimenting.

When the \* prompt appears, you must tell the Apple II to switch over to BASIC.

#### CTRL-B into BASIC

There is a Monitor command which instructs the Apple II to switch to BASIC. To issue the command, press and hold the CTRL key while you press and release the B key. Then release the CTRL key and press the RETURN key. A different prompt will immediately appear underneath the \*. Depending on the kind of Apple II you have and what options it has, you will either see the > prompt or the ] prompt.

BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code, a widely used computer language that was developed at Dartmouth University. The Apple II supports two versions of BASIC: Integer BASIC and Applesoft. Both versions contain enhancements to the original Dartmouth BASIC.

#### > is Integer BASIC

The > prompt indicates that the Apple II is ready for instructions in Integer BASIC. The rules for Integer BASIC are different from those for Applesoft, but the two

have many things in common. For now we will stick to instructions that you can use in either version, so you need not worry about which BASIC you're using.

### **] is Applesoft**

The **]** prompt tells you the Apple II is ready in Applesoft. Don't worry for now which version of BASIC you're using. When it becomes a factor, we will talk about the differences between Integer BASIC and Applesoft.

## **THE KEYBOARD**

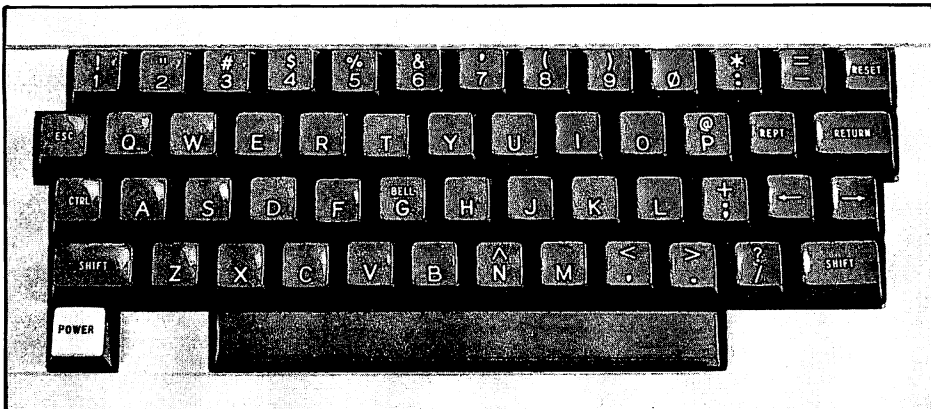
The Apple II keyboard (see Figure 2-1) looks much like the keyboard of an ordinary typewriter, but it has five keys you won't find on most typewriters. Two are on the left side, mysteriously marked **ESC** and **CTRL**. The other three are on the right, marked **RESET**, **←** and **→**. There are two more keys on the right side which you may not recognize: **RETURN** and **REPT**.

Go ahead and type on the keyboard. Nothing you can type can do any harm to the computer that can't be cured by turning the power off and on again.

As you type, you may notice that all the letters displayed on the screen are capital letters, regardless of whether or not the **SHIFT** key was pressed when you typed them. The standard Apple II only knows how to display capital letters. You can attach your TV through a special controller which enables both upper and lower case. Also, some programs know how to display both upper- and lower-case letters.

### **The RESET Key**

**RESET** is a very special key on the Apple II keyboard. When **RESET** is pressed, everything stops. No matter what the Apple II is doing when **RESET** is pressed, control of



**FIGURE 2-1. Apple II Keyboard**

the Apple II returns to the keyboard. Depending on the Apple II you are using, RESET will cause the Monitor, Integer BASIC, or Applesoft prompt to appear.

Sometimes RESET can cause a lot of problems, especially if a Disk II is in use when RESET is pressed. Therefore you must exercise extreme caution *not* to press the RESET key accidentally. Take care especially when you go after the RETURN key, as it is easy to get your finger a little too high up on the keyboard and hit the RESET key by mistake. Some versions of the Apple II guard against this hazard by insisting you use CTRL-RESET instead of just RESET (see the discussion of the CTRL key below).

### The RETURN Key

As you type along, the characters you type show up on the display screen. In addition, the Apple II saves everything you type in its memory but does not try to interpret what you type as an instruction until you press the RETURN key. The RETURN key signals the Apple II that you have finished the line you have been typing. When you press RETURN, the Apple II erases any stray characters that might be to the right of the cursor. Then it examines everything on the line that you just typed in. If those characters make up an instruction that the Apple II can understand, it will take the appropriate action. Otherwise you will hear a beep and see one of these messages (or possibly some other message):

```
?SYNTAX ERROR
*** SYNTAX ERR
```

The Apple II is letting you know that it did not understand what you meant by the characters you typed before you pressed RETURN. You must retype the line (without the error that tripped up the Apple II).

### The SHIFT Key

Letters are always upper-case on the standard Apple II, so the SHIFT key does not toggle between lower-case and upper-case letters. It does allow some keys to produce two different characters. You get one character by pressing a key with the SHIFT key held down and another by pressing the same key without holding the SHIFT key down. The character you get with the SHIFT key is printed on the top edge of the key.

We use the notation SHIFT- to describe a double keystroke involving the SHIFT key. For example, SHIFT-3 produces the # character.

Some letter keys do not display letters when they are typed with the SHIFT key held down. The N key, for example, will display ^, and @ is displayed when you type a SHIFT-P. Although the G key has the word BELL on it, SHIFT-G does not display a bell, just a G.

### The CTRL Key

CTRL is a contraction of control. The CTRL key is always used together with another key in the same manner as the SHIFT key. You hold the CTRL key down

while you press and release another key. This book designates the use of the `CTRL` key in conjunction with another key by prefixing the name of the other key with `CTRL-`. For example, `CTRL-B` means press the `CTRL` and `B` keys simultaneously.

The `CTRL` key, like the `SHIFT` key, allows some keys to have an additional function. The `G` key is the only key labeled with its `CTRL` function: `BELL`. The meaning of `BELL` becomes obvious when you type `CTRL-G` and the Apple II responds with a beep. You are ringing the Apple II bell.

There are other `CTRL` combinations which cause various reactions from the Apple II. We recently covered the Monitor command `CTRL-B`, which puts the Apple II into `BASIC`.

Another handy combination is `CTRL-X`. It tells the Apple II to disregard everything you've typed on the current display line; you want to start over. Try typing something and then type `CTRL-X`. The cursor zips back to the beginning of the line. The old characters still appear on the display screen but the Apple II has no recollection of them whatever. As far as it's concerned, you never typed them.

### The Esc Key

`Esc` stands for escape, which is a term left over from the days when teletypes were common computer terminals. Somehow the name has stuck, even though pressing the key causes no breakout. The `Esc` key has a variety of uses, some of which come up in this chapter and the balance appear in the next chapter.

Unlike the `SHIFT` and `CTRL` keys, the `Esc` key is never used by holding it down while pressing another key. `Esc` is always pressed and released before the next key is pressed and released. This two-key operation is called an *escape sequence*.

A simple escape sequence is `Esc-@`. You type this escape sequence by first pressing and releasing the `Esc` key, then pressing and holding the `SHIFT` key while you press and release the `P` key (`SHIFT-P = @`). The result? Everything on the screen is erased, and the cursor moves to the upper left corner. (In computer jargon, the upper left corner of the display screen is called *home*.)

### The ← and → Keys

The two arrow keys are called *left-arrow* and *right-arrow*.

You will find the ← and → keys very useful because they allow you to correct any typing mistakes you might make, and allow you to change information you have already entered. The ← key works like the backspace key on a typewriter. Each time you press it, the character under the cursor is erased from the Apple II memory and the cursor backs up one space. Try it right now. Type in any word (try `PRINT`). Press the ← key several times and watch the cursor back up along the word you just typed in. Notice that the characters you back over do not disappear from the display screen. You can rest assured the Apple II has put them out of its memory. Try backing the cursor all the way to the left edge of the screen. When you get to the edge and press the ← key again, the cursor jumps down one line and a new prompt character appears.

As you might suspect, the → key moves the cursor to the right along the display line. As the cursor moves forward along the line, every character it passes over gets copied into the Apple II memory exactly as though you had pressed the key to generate that character. To see the → key in action, type in another word and back the cursor up a few spaces using the ← key. Now press the → key a few times. Each time you press this key, the character the cursor passes over gets put back into the Apple II memory exactly the same as if you had retyped it.

### **The REPT Key**

REPT is short for repeat. If you hold the REPT key and any other key down at the same time, the other key will be repeated until you release one or both keys. This is especially handy when the other key is an arrow key and you have a lot of characters to erase (←) or recopy (→).

For the repeat feature to work right, you must first hold down the key you wish repeated and then press the REPT key. Release REPT and the repeating stops.

### **The Other Keys**

The other keys on the Apple II keyboard are no doubt familiar to you. These are the letters of the alphabet, the numbers zero through nine, and a standard set of symbols.

Many typists do not distinguish between the number zero and the letter O or the number 1 and the lower-case letter l. The Apple II can't cope with this ambiguity. You must be very careful to type a numeral when you mean a numeral. To help you remember, the Apple II keyboard shows the zero with a slash through it, and zeros are displayed on the screen with that slash, too.

## **THE CASSETTE RECORDER**

If your Apple system includes a cassette recorder, you can load programs from cassette tapes. Some program tapes come with the Apple II, you can buy others, and you can make your own as well (we'll tell you how in Chapter 4).

### **Handling Cassettes**

You should exercise care with the cassettes themselves. They are very easily damaged, and not easily replaced.

Be very careful not to touch the surface of the tape in the cassette. No matter how clean your skin is, natural oils will contaminate the tape. Make sure you put tapes back in their cases when they are not being used. Never store them in hot areas, direct sunlight, or near magnetic fields (like those found near electric motors).

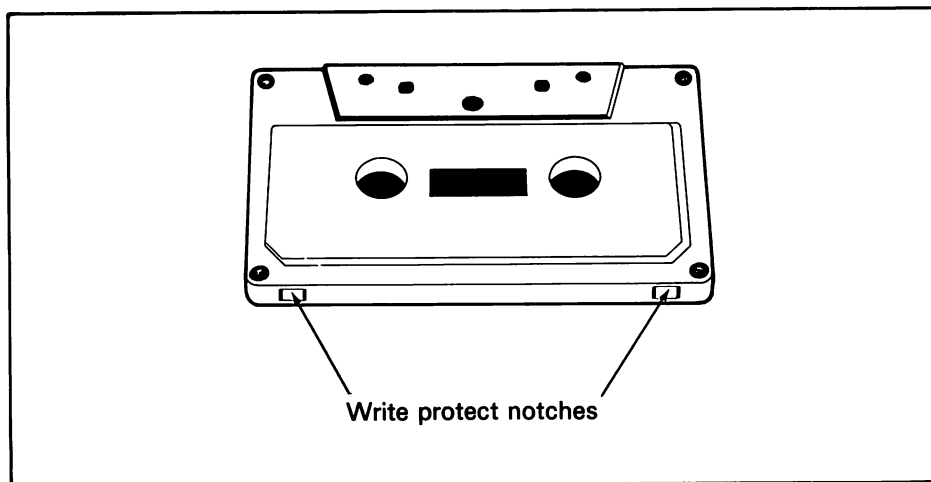


FIGURE 2-2. Cassette Write-Protect Notches

### Label Every Cassette

You should label every cassette with information about the programs it contains. This avoids the headache of searching through cassette after cassette for the program you need.

### Write-Protecting Cassettes

Each cassette has two notches in the rear edge. Most cassette recorders can sense the notches and will not record when they are present. Blank cassettes have tabs covering the notches so the tape may be recorded over. You can protect important programs by removing the correct tab and exposing the notch.

To determine which tab is correct, hold the cassette so that the exposed tape is away from you and the side you wish to protect is facing up. Remove the tab on the right side to prevent recording over the side facing up (see Figure 2-2). Covering a notch with adhesive tape will allow recording over a cassette that has been protected.

### Adjusting the Playback Volume

You must set the volume control on your cassette recorder at the proper level for the Apple II. If the volume is too low or too high, the information of the tape will be distorted and the Apple II will not be able to understand it.

The only way to determine what volume level is correct for your tape recorder is by trial and error. Here is the general procedure. First, try to load a program with the volume control set very low. If the low setting does not work, set the volume a little higher and try again. Keep adjusting the volume upward until you can successfully load the program.

You can use one of the tapes that come with the Apple II. If the prompt character next to the flashing cursor is ], find the cassette labeled "Color Demosoft." If the prompt is >, locate the tape labeled "Color Graphics."

Insert the cassette into the recorder. Be sure that the program label faces up.

For each position of the volume control:

1. Rewind the tape completely.
2. On the Apple II keyboard, type the word LOAD.
3. Depress the PLAY button on the cassette recorder to start the tape.
4. Press the RETURN key.

After you press RETURN the cursor will disappear. After 15 or 20 seconds you can analyze your success.

If you get the message ?SYNTAX ERROR or \*\*\*SYNTAX ERR, do not adjust the volume, just go back to Step 1 and try again. If this keeps happening, try cleaning the cassette recorder heads, or use a different tape.

If nothing happens, or if ERR or ERRERR is displayed, press RESET, set the volume a little higher, and try again.

If you hear a beep and no message appears, things are going well. The Apple has found the beginning of the program on the tape and is loading it. After about 15 more seconds (depending on the length of the program on the tape), there will be another beep and the prompt and cursor will reappear on the screen. You can now stop the tape. Make a note of the volume setting so you don't have to repeat this procedure after using the recorder away from the Apple II.

When you see the BASIC prompt and cursor, the program has been successfully loaded.

## USING THE DISK II

If you have one or more Disk II drives connected to your Apple II, you can get programs on diskettes instead of cassettes. The "System Master Diskette" supplied with the Disk II has most of the programs Apple Computer Inc. provides on cassettes, plus a few extras specially designed for the Disk II.

### Handling Diskettes

You must be very careful when you handle a diskette. Diskettes are much more delicate than cassette tapes. *Never* bend a diskette. *Never* touch the surface of the diskette (the part inside the oval shaped hole), and *never* force a diskette into the Disk II drive. Always replace diskettes in their envelopes when you remove them from the drive, and protect them from heat, direct sunlight, and magnetic fields (like those found near electric motors). Be especially careful with the "System Master Diskette."

### How to Insert Diskettes Into the Disk II Drive

The proper way to insert a diskette into the Disk II drive is shown in Figure 2-3. Hold the diskette between your thumb and forefinger so that your thumb covers the label. Open the door on the Disk II drive and gently slide the diskette all the way into the drive. There should be almost no resistance. If the diskette will not go in easily, remove it and try again. Make sure you are holding the diskette as level as possible. Once the diskette is inside the drive, gently close the drive door. The door should close very easily. If there is any resistance, release the door and push the diskette completely into the drive, then try again. If you force the door shut you will destroy the diskette. Sometimes it helps center the diskette if you wait to close the door until after the Disk II starts spinning.

### THE DISK OPERATING SYSTEM

Before you can use any disk drive, a special program called *Disk Operating System* must be in memory. The Disk Operating System, or *DOS* for short, is a special program which controls all the disk related activities. The process of placing a copy of DOS in memory is call *booting*. In computer jargon you can say "boot the disk" or "boot the DOS" or just "boot DOS."

You must boot DOS each time you turn the Apple II off and back on again.



FIGURE 2-3. Inserting a Diskette into the Disk II



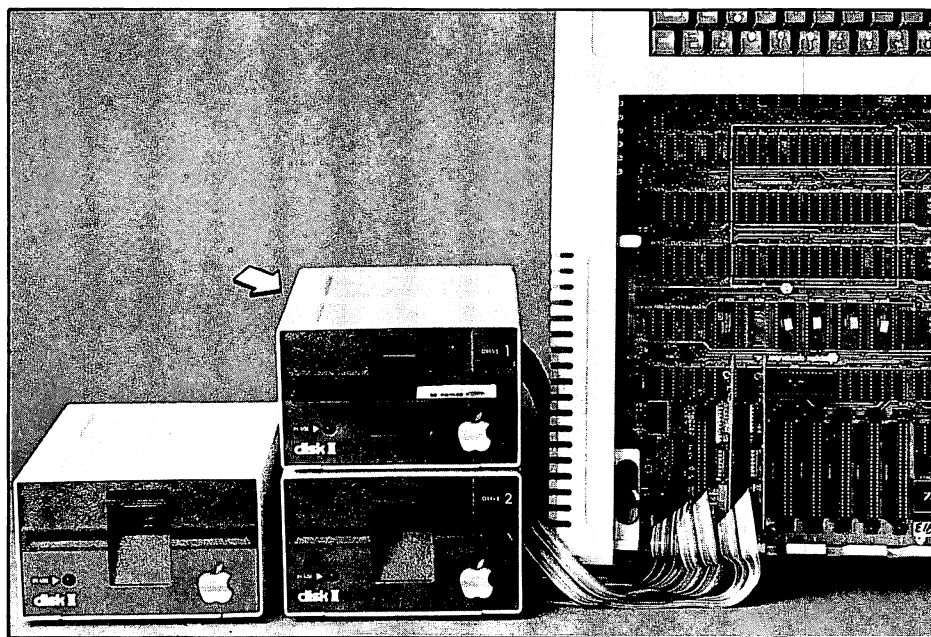


FIGURE 2-4. Standard Disk Drive for Booting DOS

### Booting DOS

There are several different ways to boot DOS, depending on the configuration of your computer and the language being used to initiate the boot. Each method assumes the disk drive is plugged into slot number 6 and connected to the drive 1 terminal of the controller card. This is shown in Figure 2-4.

Insert the "System Master Diskette" into the correct drive and close the drive door. In some cases when the Language System is present, you must use a special diskette, the "Integer and Applesoft II" diskette, before the DOS disk. This situation is described below in the section on booting with the Language System.

After a successful boot using one of the methods described below, the screen should look like one of those shown in Figure 2-5.

### Autostart Booting

The easiest way to boot the disk is autostart booting. As the name implies, booting is automatic. In order for autostart booting to be possible your computer must have an Autostart Monitor. If, when you turn your Apple II on, the Disk II drive makes clicking and whirring sounds, and the red IN USE lamp on the front of the drive cabinet lights up, you have an Autostart Monitor.

When the Autostart Monitor is present without the Language System, booting DOS is a one-step procedure. With the Apple II power switched off, place the

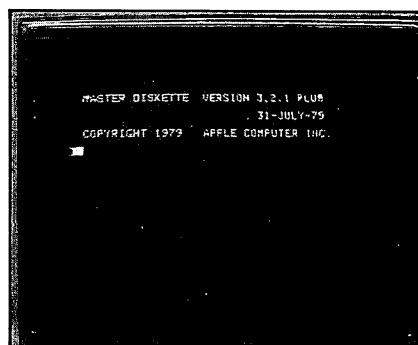
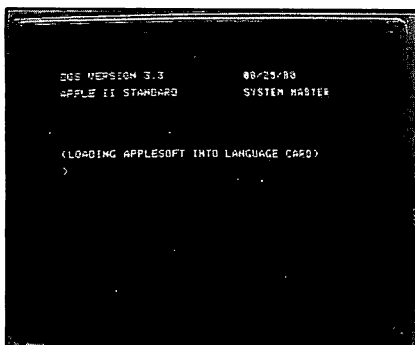
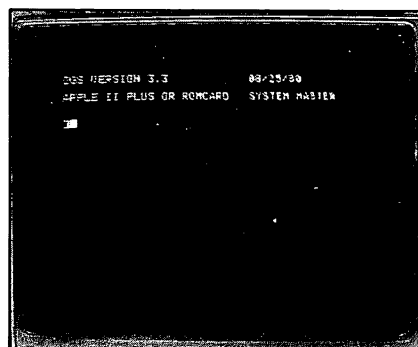
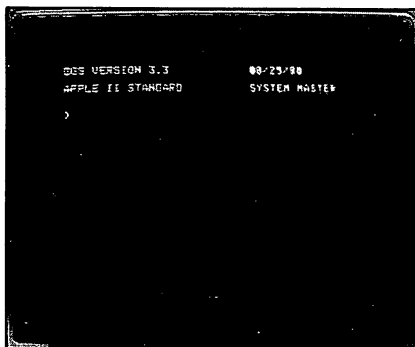
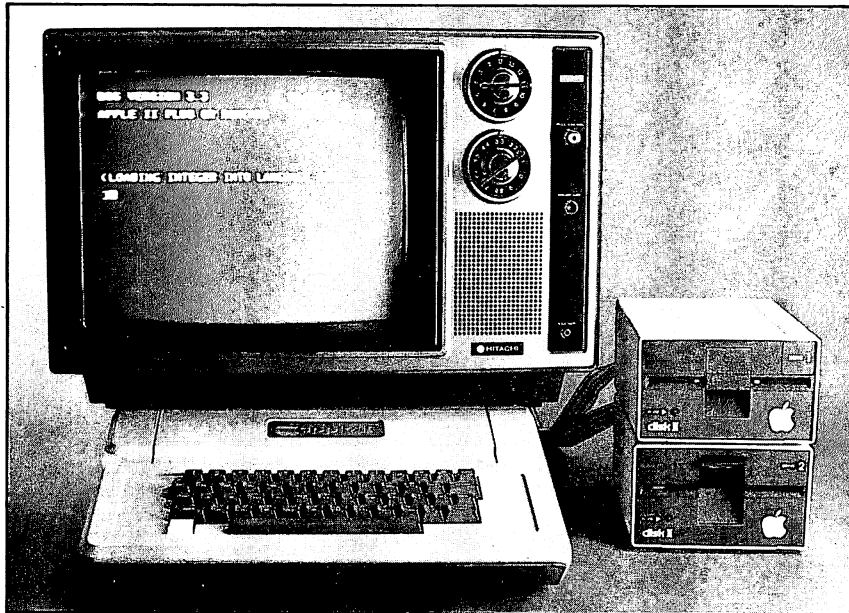


FIGURE 2-5. Successful Boot of System Master Diskette

“System Master Diskette” in the proper Disk II and close the drive door. Turn the power on. After a few seconds, the disk will stop and the screen will look like one of those in Figure 2-5.

### Booting from the Monitor

When the \* prompt character appears on the screen, the Assembly Language Monitor is waiting to accept commands. There are a couple of ways to boot the disk from the Monitor.

#### Monitor Jump Booting

You can boot DOS from drive 1 of slot 6 (see Figure 2-4) with the following Monitor command:

\*C600G

Remember to press RETURN. The red IN USE lamp on the Disk II will light up, and after a few seconds the screen will look like one in Figure 2-5.

#### CTRL-P Monitor Boot

The other Monitor command you can use to boot the DOS is CTRL-P. To boot from the Monitor using this command, type the slot number of the drive you wish to boot from (usually 6), then type CTRL-P (nothing will appear on the screen). Now press RETURN. After a few seconds the screen will look like one of those in Figure 2-5.

### Booting from Integer BASIC or Applesoft

The same boot commands are recognized by both Integer BASIC and Applesoft.

After the BASIC prompt character (> in Integer BASIC, or ] in Applesoft) type the letters IN or PR, then a pound sign (#), and then the slot number of the drive you wish to boot from (usually drive 6). The command should look like one of these:

IN#6

PR#6

Now press the RETURN key. After a few seconds the screen will look like one of those in Figure 2-5.

### Booting with the Language System

Under some conditions, booting DOS with the Language System present is a two-step procedure. The version of DOS being booted determines the procedure.

The version number of DOS on the "System Master Diskette" is stamped on the diskette label. It is a number like 3.3, 3.2.1, or 3.2.

Booting version 3.3 of DOS is virtually the same with or without the Language System. Use the "System Master Diskette" with any one of the procedures described above. DOS is booted from the diskette into memory. Another step occurs automatically which makes both Integer BASIC and Applesoft available immediately, via the Language System. When the Disk II stops, booting is finished, and the screen is similar to one of those in Figure 2-5.

Booting DOS versions 3.2.1, 3.2, and lower requires two diskettes. First you must place the diskette labeled "Integer and Applesoft II" in the correct Disk II, usually the one connected to slot 6, drive 1. Then perform any of the boot procedures described above. The drive will make sounds and the red IN USE lamp on the front of the drive cabinet will light up. In a few seconds, you will see this message on the display screen:

```
INSERT BASIC DISK AND PRESS RETURN
```

Now open the drive door and remove the diskette. Insert the "System Master Diskette" in the drive and close the door. Press the RETURN key on the keyboard. After a few more seconds, the screen will look like one of those in Figure 2-5. DOS is now successfully booted.

### How to See the Diskette Catalog

If you have successfully booted the "System Master Diskette," you may be interested in knowing what programs that diskette contains. On the Apple II keyboard, type:

```
CATALOG
```

The screen display should look something like the one shown below. (Did you remember to press the RETURN key?)

```
DISK VOLUME 254
*I 002 HELLO
*I 053 APPLE-TREK
*I 018 ANIMALS
*B 009 UPDATE 3.2.1
*I 014 COPY
*I 009 COLOR DEMO
*I 053 BRICK OUT
*I 026 SPACE WAR
*I 050 THE INFINITE NO. OF MONKEYS
*I 051 COLOR SKETCH
*I 053 SUPERMATH
*I 026 APPLEVISION
*I 017 BIORHYTHM
*I 027 PINBALL
```

```
NEW
10  REM INITIALIZED ON date
20  REM SYSTEM MEMORY bytes
30  REM DOS VERSION number
40  PRINT "disk name"
50  END
```

FIGURE 2-6. Model Greeting Program

### Booting Other Diskettes

We just explained how to boot DOS from the "System Master Diskette." The procedure is exactly the same for any diskette that has been set up for the Disk II. Diskettes set up for other computers or other kinds of disk drives usually will not work in the Disk II drive.

### PREPARING BLANK DISKETTES

From time to time you may need extra diskettes for the programs you run on your Apple II. Before you can use a diskette with the Disk II for the first time, you must initialize it. If the application program you are using includes specific instructions for initializing diskettes, by all means use them. In their absence, you can use the following general instructions for preparing extra diskettes.

The initialization process gets a diskette ready for subsequent use. During initialization, whatever BASIC program is currently in the Apple II memory is saved on the diskette and becomes that diskette's *greeting* program. The greeting program is automatically loaded and run whenever you boot the diskette. If you have a greeting program in mind, fine. If not, you can use Figure 2-6 as a model. Use the program exactly as shown except wherever there are italics, replace them with actual information. The *date* is the date on which you initialize the diskette. The *bytes* are the number of bytes in the system memory (32K, 36K, etc.). The *number* is the version number of DOS that is present on the diskette most recently booted (something like 3.2, 3.2.1, or 3.3). The *diskette name* is any name you want to use for that diskette. Illustrated below is an actual example of the initialization program:

```
NEW
10  REM INITIALIZED ON 1/1/81
20  REM SYSTEM MEMORY 48K
30  REM DOS VERSION 3.2.1
40  PRINT "MISC., VOL. 3"
50  END
```

So to initialize a diskette, first prepare the greeting program, put the diskette in the drive, and type the following command:

```
INIT HELLO
```

Make sure the drive door is shut, and press RETURN. The red lamp on the drive lights up, accompanied by the usual whirring and clicking sounds. The entire initialization process takes about two minutes, so be patient. When the lamp goes out, initialization is complete.

Now prepare a label for the new diskette. Remove the diskette from the drive and apply the label.

## LOADING AND RUNNING A PROGRAM

There are many programs already written for the Apple II. Some come on cassette, some on disk, and some on both. Some programs are written in Integer BASIC while others use Applesoft. Generally speaking, an Applesoft program will not work with the Integer BASIC prompt, and Integer BASIC programs will not work with the Applesoft prompt.

### USE THE RIGHT VERSION OF BASIC

While most versions of the Apple II have both kinds of BASIC, not all do. The Apple II Plus does not have Integer BASIC unless the Language System card or Integer BASIC card is present. On the other hand, Applesoft is not available on many versions of the Apple II until you load its interpreter from cassette or disk.

If your Apple II system has the Language System or the Applesoft Firmware card, it will automatically use the right version of BASIC for any application program you load. Otherwise, you must be sure the correct prompt character is in effect when you load a program from cassette or disk.

On a standard Apple II it is easy to get the Integer BASIC prompt (>). Press the RESET key, and then type CTRL-B (remember to end with RETURN).

It is harder to get the Applesoft prompt (}) on a standard Apple II since it resides on either cassette or disk. You must instruct the Apple II to load the Applesoft interpreter into its memory from the disk or cassette.

Before you can use the disk drive you must boot DOS as described earlier in this chapter. Once DOS is booted, you can load the Applesoft interpreter by placing the System Master Diskette in the Disk II and typing the following command:

```
FP
```

The drive will click and whirl for a few seconds and then the Applesoft prompt character (}) will appear on the display screen.

You can also get the Applesoft interpreter from cassette tape. Use the cassette

labeled "Applesoft II" (from Apple Computer Inc.). Put it in the recorder and rewind it all the way. Type the command:

LOAD

Before you press RETURN, depress the PLAY button to start the tape. Soon you will hear the Apple II beep, which means it has started to load the Applesoft interpreter. It takes about one and one-half or two minutes to load the Applesoft interpreter from a cassette tape. Once it is loaded, you will hear a second beep. Stop the cassette recorder. The Applesoft prompt will be on the display screen.

From Applesoft, you can switch to Integer BASIC by typing the command:

INT

If you then wish to switch back to Applesoft, you must reload the Applesoft interpreter from disk or cassette.

## LOADING A PROGRAM FROM CASSETTE

With the proper version of BASIC selected (or your Apple II able to make the selection automatically), these are the steps for loading a program from cassette:

1. Position the tape to the start of the program. This will usually be the beginning of the tape, in which case you must rewind the tape completely. If the program you want is not the first program on a cassette tape, you must load in turn each program that precedes it on the tape. Repeat the following steps for each extra program you must pass over.
2. On the Apple II keyboard type the command:

LOAD

3. Depress the PLAY button on the cassette recorder to start the tape.
4. Press the RETURN key.

After you press RETURN, the cursor will disappear. After a few seconds, the Apple II will beep to signal that it has started to load the program. Some time later it will beep again signaling that it has finished loading the program. Use the STOP button on the cassette recorder to stop the tape.

The program is now loaded. If you hear no beeps or if you get any error messages during the load process, recheck the volume control adjustments according to the directions given earlier in this chapter. If you still have problems, the cassette tape you are using has probably gone bad and you will have to replace it.

## LOADING A PROGRAM FROM DISK

Once DOS is booted, you can load a program from a disk with a command like this:

LOAD *program name*

In actual use, *program name* is the name of the program to load. Naturally, a program by that name must be present on the disk that is in the drive.

## STARTING A PROGRAM RUNNING

When the program you want is loaded, use the following command to get it started:

RUN

The program takes over control of the Apple II, including the keyboard and display screen. To regain control, you can type CTRL-C in most programs. You may have to press RETURN as well. If this does not work, check the specific operating instructions for the program you are using. In a dire emergency, you can press the RESET key or turn the Apple II power off and back on again, but in either case you may lose the program.

## SETTING TV COLOR

The Apple II features full color graphics. If any of the programs you plan to use or write will use this feature, you should adjust the color settings on your television set or TV monitor for the correct balance. Apple Computer Inc. provides a program to assist in this task. With the Integer BASIC prompt (>), use the "Color Graphics" cassette tape. With the Applesoft prompt (}), use the "Color Demosoft" cassette tape or the COLOR DEMOSOFT program on diskette. In either case, load and run the program according to the instructions given earlier in this chapter.

The screen will look like the one shown in Figure 2-7.

What you see in Figure 2-7 is called a *menu*. A menu lists several choices (in this case, four) and asks you to choose one. To set the TV color, select number one by pressing the 1 key and then the RETURN key. Now the screen will show 16 vertical stripes of color like the photograph in Figure 2-8, except in color.

As you see, the colors are named along the bottom of the screen. From left to right, the colors are:

BLAK	Black	BROWN	Brown
MGTA	Magenta	ORNG	Orange
DBLU	Dark Blue	GREY	Grey
PURP	Purple	PINK	Pink
DGRN	Dark Green	LGRN	Light Green
GREY	Grey	YELO	Yellow
MBLU	Medium Blue	AQUA	Aqua
LBLU	Light Blue	WITE	White

Now adjust the contrast, brightness, color, and tint controls on the TV until you get acceptable colors that agree with their names. Pay particular attention to purple, pink, yellow, and the three blue colors.

When you have finished adjusting the color, press the RETURN key and the



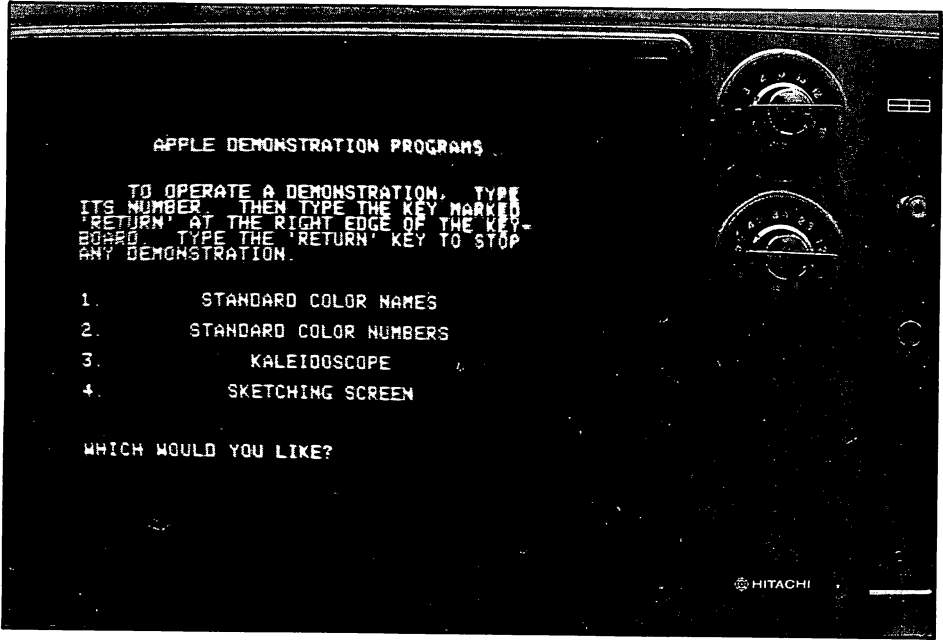


FIGURE 2-7. COLOR DEMOSOFT Program Menu

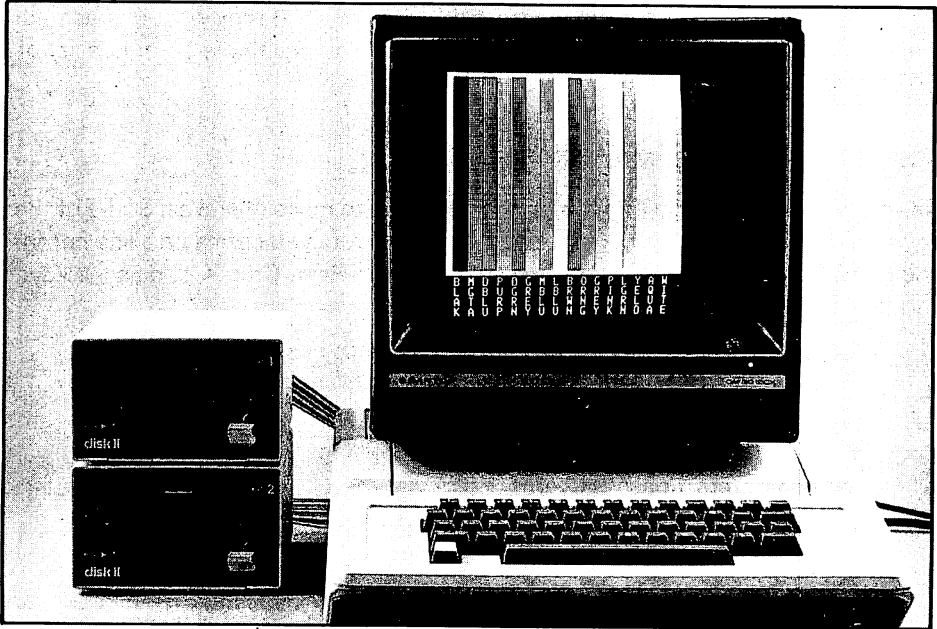


FIGURE 2-8. Adjusting TV Color

menu reappears. You can choose any item on the menu if you wish to experiment a bit. To end the program, type **CTRL-C** (you'll have to end with the **RETURN** key).

## MISCELLANEOUS COMPONENTS

This concludes the specific operating instructions for system components. For other components in your system, use the operating manual for that component. If you have a printer, for example, then you must refer to the printer manual, since operating instructions for printers vary widely from one brand to another.

## COPING WITH ERRORS

The Apple II is a marvelous piece of equipment but it shares a problem common to all computer systems. It lacks imagination. Every instruction you give must be exactly right or it will not work as you expected. The results of a mistake can run the gamut from annoying to aggravating to devastating.

### ERROR MESSAGES

When you type something in wrong and press **RETURN**, the Apple II usually responds with a beep and a somewhat cryptic *error message*. Often the message gives you a clue as to what you did wrong; then again it may not. The general remedy is the same in either case: retype the line. You will find a complete list of error messages in Appendix C.

### CORRECTING TYPING MISTAKES

As you type on the Apple II keyboard you're bound to make mistakes. Some of the keys we described earlier make it easy to correct errors you notice on a line before you press **RETURN** to end the line. They are the **←**, **→**, **REPT**, **CTRL-X**, and **Esc-@** keys and key sequences.

- The **←** key backspaces the cursor and erases characters it passes over. Characters are erased from the program line even though they still appear on the display screen.
- The **→** key moves the cursor forward, copying over (retyping) characters it passes over.
- The **REPT** key, used in conjunction with the **←** and **→** keys, enables fast-backspacing and fast-forwarding.
- The **CTRL-X** command cancels the line you're currently typing.
- The **Esc-@** command clears the display screen and leaves the cursor in the upper left corner.

Let's see how you might use these editing features. Suppose you want to type this:

```
LOAD COLOR DEMOSOFT
```

but instead you get this far and notice you've made an error:

```
LOAS COLOR DEMOSOFT
```

You have two choices. You can type CTRL-X to cancel the line and start all over again or you can use the ← key to back up and correct the mistake. Press and hold the ← key. Then press the REPT key. The cursor races back to the start of the line. Take your finger off the REPT key when the cursor gets to the first error. Remember that as it backs up, the cursor is wiping every character it passes from the computer memory. If you back too far, use the → to line the cursor up over the offending S. Now press the D key, and presto! The line *reads* correctly. Do you know what will happen if you press RETURN right now?

```
LOAD COLOR DEMOSOFT
```

All the characters to the right of the cursor will vanish.

```
LOAD
```

That's not right! Before you press RETURN, you must move the cursor to the end of the line. You could retype the rest of the line, but you might make another mistake. The → key (together with the REPT key) moves the cursor to the end of the line, automatically retyping the characters it passes.

## ACCIDENTAL RESET

Sooner or later you will hit the RESET key when you did not intend to. It is inevitable, unless your Apple II requires you to type CTRL-RESET instead of just plain RESET. You can reduce the hazard by carefully prying off the plastic keytop, leaving just the keyswitch shaft available (Figure 2-9).

You can take this one step further and make it physically harder to push down the RESET key. Get a small rubber washer about 3/8 inches inner diameter, 1/2 inch outer diameter, and 1/8 inch thick. Remove the RESET keytop as shown in Figure 2-9. Work the washer over the exposed square flange of the keyshaft as shown in Figure 2-10. Replace the keytop.

## Recovering from RESET

What you can do about an accidental RESET depends on what you were doing at the time the RESET occurred, and what kind of Apple II you have.

Any program you use on the Apple II should have specific instructions about what to do if you accidentally press the RESET key while running that program. Be sure you know what to do before you start your program. If you press RESET while running a BASIC program you *will* be able to restart the program from the beginning. This is small consolation during some phases of accounting applications and the like, since rerunning could foul things up.



FIGURE 2-9. Guarding Against Accidental RESET: Phase I



FIGURE 2-10. Guarding Against Accidental RESET: Phase II

When you press **RESET**, the Apple II stops everything it was doing. Control returns to the keyboard; you will see a prompt character and the flashing cursor on the bottom of the display screen. If the prompt is a BASIC prompt then you can restart any program you were running by typing the **RUN** command. Some of what you did with the program since the last **RUN** command will probably be lost.

If you see a Monitor prompt character (\*) after pressing **RESET** then you can switch back to BASIC by typing **CTRL-C** *unless* you were using cassette-based or disk-based Applesoft. To switch from the Monitor to disk-based Applesoft, type the following Monitor command:

**\*3DOG**

To switch from the Monitor to cassette-based Applesoft, type the following Monitor command:

**\*OG**

**CAUTION:** Be sure you use the correct command. If you are not sure which to use, ask someone who is. Use the wrong command and you are dead. You will have to reload your program, Applesoft, and possibly even DOS.



# 3

## Programming in BASIC

This chapter teaches you how to start writing your own BASIC programs on the Apple II.

BASIC is a programming language. Like any programming language, it consists of a set of statements, which you combine to create programs. A program defines the task you want the computer to perform.

We could teach you BASIC by forcing you first to learn BASIC statements, one by one. But you would probably give up, since individual statements are not very meaningful. A study of individual BASIC statements quickly degenerates into learning a bunch of arbitrary rules that tell you nothing about programming or good programming practice. Therefore rigorous definitions of all BASIC statements have been relegated to Chapter 8. Look up individual statements in Chapter 8 when you need to, but do not try to read Chapter 8 before you read this chapter.

### STARTING UP BASIC

There are two different versions of BASIC available on the Apple II. Some models of the Apple II have Integer BASIC, some have Applesoft, and many have both. For now, you don't need to worry about whether you are using Integer BASIC or Applesoft. Later, when the difference becomes significant, we will discuss it in more detail.

### Start with a BASIC Prompt Character

The Apple II is a multilingual computer. If you wish to program it in BASIC, it must be expecting instructions in that language. You will find complete instructions for starting up BASIC in Chapter 2. Let's quickly review.

On an Apple II with the Autostart Monitor, you need only turn the machine on and press RESET (CTRL-RESET on some versions of the Apple II keyboard).

If your Apple II does not have the Autostart Monitor, turn it on and press CTRL-B, then RETURN.

The prompt character in the left margin, next to the cursor, tells you which BASIC you are in. You have Integer BASIC if the prompt is a ">". The prompt "[" means you have Applesoft.

## IMMEDIATE AND PROGRAMMED MODES

Before we start worrying about how to switch the Apple II between Integer BASIC and Applesoft, let's take a look at some simple BASIC statements you can use in either version of BASIC. Some of the examples in this chapter use the Integer BASIC prompt (>) while others use the Applesoft prompt ([ ]). You can try the examples in either version of BASIC, no matter which prompt you see in the book, *unless* specifically stated otherwise. Those examples with neither prompt can be used in either version of BASIC.

### PRINTING CHARACTERS

When you first put the Apple II in BASIC, it is in *immediate mode*, also called direct or calculator mode. In this mode, the computer responds immediately to any instructions you issue it. Try typing in this example:

```
PRINT "LET SLEEPING DOGS LIE"
```

Don't forget to press the RETURN key after the last quotation mark. The Apple II prints the following message:

```
LET SLEEPING DOGS LIE
```

If the Apple II prints the message ?SYNTAX ERROR or \*\*\* SYNTAX ERR, it is telling you your command was indecipherable. You probably misspelled the word PRINT. If the Apple II just prints the number 0 instead of any message, it means you left out the first quotation mark. In either case, you can just type the instruction in again, but be more careful this time! Computers are extremely particular about spelling and punctuation. Even the slightest error can cause the computer to balk or — even worse — to do the wrong thing.

A command like the one above instructs the computer to print everything between the quotation marks onto the display screen.



There is a limit to the length of the message you can put between quotation marks. The limit is different for Integer BASIC and Applesoft, but in both cases it exceeds the width of the display screen. This means a command can occupy more than one display line. Long commands like this automatically *wrap around* to the next lower line on the display screen. Try this command:

```
PRINT "UNDER NORMAL CIRCUMSTANCES, THE M
      AN WOULD BE CONSIDERED CRAZY"
      UNDER NORMAL CIRCUMSTANCES, THE MAN WOULD
      BE CONSIDERED CRAZY
```

Integer BASIC allows about 120 characters. If you exceed the limit, you will get the message **\*\*\* TOO LONG ERR** after you press RETURN.

Applesoft allows 255 characters. As you approach the limit, the Apple II starts beeping. When you exceed the limit, it prints a backslash (\) and automatically cancels your entry, as if you had pressed CTRL-X.

## PRINTING CALCULATIONS

You can use the Apple II in immediate mode as you would a calculator; it responds directly with the answers to arithmetic calculations. Try these examples:

PRINT 4+6 10	Addition
PRINT 500-437 63	Subtraction
PRINT 100*23 2300	Multiplication
PRINT 96/12 8	Division
PRINT 3^2 9	Exponentiation
PRINT 3*4*10-800 -680	Combination

The correct answers are on the line immediately following each of the commands. Notice that you do not use quotation marks in these examples. Do you know what would happen if you did? Try it and see if you're not sure.

Integer BASIC has a maximum and minimum limit on the value of a calculation. If the value of a calculation is more than 32767 at any point during the calculation, the error message **\*\*\* >32767 ERR** appears. If the value is less than -32767, the message **- \*\*\* >32767 ERR** appears. Some examples of these errors are shown below. The last example shows division by 0.

```
>PRINT -32766-2
-*** >32767 ERR
```

```
>PRINT 2^15-1
*** >32767 ERR
>PRINT10/0
*** >32767 ERR
```

Decimal fractions are not allowed in Integer BASIC. So, if you perform a division calculation that does not come out even, the remainder will be discarded. For example, try this calculation:

```
>PRINT 9/2
4
```

Applesoft does allow fractions. Numeric values can have a total of nine *significant digits* including both fractional and nonfractional parts. This means that values with more than nine digits are rounded off to nine or fewer nonzero digits. These examples illustrate how this works:

```
JPRINT 12.34567896   Rounded Up
12.345679

JPRINT 12.34567894   Rounded Down
12.3456789
```

If you try some of your own arithmetic calculations in immediate mode in Applesoft, you will notice the result is sometimes displayed using scientific notation.

```
JPRINT 123456789123
1.23456789E+11
```

If you do not understand scientific notation, stick to simple calculations for now. We will talk more about scientific notation and numeric values later in this chapter.

### Abbreviated PRINT Statement

Applesoft allows you to abbreviate the PRINT statement with a dollar sign (\$). Here are some examples you can try:

```
J?"TIME MARCHES ON"
TIME MARCHES ON

J?13-46*6
-263
```

### ERROR MESSAGES

So far we have mentioned several messages the Apple II will issue when it detects situations it cannot cope with. It beeps to draw your attention to the fact that something is wrong, and it tries to diagnose the problem for you. Its diagnostic abilities are limited, though, so do not expect a definitive analysis of your error. There are fewer than 35 possible messages the Apple II can use to respond to the myriad of possible errors and combinations of errors.

Throughout this chapter and the rest of the book we will make note of some of the errors that can occur in specific situations, especially the insidious ones. All error messages are listed alphabetically in Appendix C.

### Error Message Format

Error messages have a slightly different format in Integer BASIC than in Applesoft, as shown below:

```
JPRNIT "THE LAVA FLOWS"

?SYNTAX ERROR
]

PRNIT "THE LAVA FLOWS"
*** SYNTAX ERR
>
```

### EXTRA BLANK SPACES

Are you struggling with the question of where to put spaces in a line and where not to? Don't worry. The Apple II interprets a BASIC line by the elements in it. Spaces, or blanks, are irrelevant. The only place you must put spaces is within quotation marks, where you want spaces to be part of a message.

### STATEMENTS, LINES AND PROGRAMS

A program consists of one or more statements which provide the Apple II with an exact and complete definition of the task which it is to perform. If the task is short and simple, the program can be short and simple as well. The immediate mode instructions we have experimented with so far are each small, simple programs. Each one has just one statement — one instruction to the Apple II. These are trivial cases. Most programs have 10, 100, 1000, or even more statements. Consider the following statements:

```
PRINT "COWS MOO"
COWS MOO

PRINT "FOR FANCY BLUE"
FOR FANCY BLUE

PRINT "HOOF-B-NU"
HOOF-B-NU
```

Each of these immediate mode programs prints a line of text on the display screen. Each program has exactly one statement and exactly one line.

Applesoft allows you to put more than one statement on a line. You separate multiple statements on the same line with a colon (:). Compare this immediate



The program line is conveniently left at the top of the screen. This is because the program displays just enough lines to scroll the program line to the top of the a 40-column screen, but not off it.

When the one-line program described above is finished, the Applesoft prompt is displayed with the cursor adjacent to it on the bottom line of the display screen.

## PROGRAMMED MODE

The programming we have done so far is educational and, hopefully, interesting, but there is only so much you can do in immediate mode. This is especially true in Integer BASIC, with its limit of one statement per program line. Another problem with immediate mode programs is that you have to retype the program each time you want to use it. There are some advanced editing techniques which we will discuss shortly that will allow you to reuse the program as long as it still appears on the display screen, but this is still a limitation.

What you need is a way to enter several program lines and to hold off using those lines. That way you can write programs to do tasks that are too complex for one-line programs.

There is a way to get around the problems of immediate mode, and that is to write programs in *programmed mode*, also called deferred or indirect mode. In programmed mode, the computer accepts and stores the program in its memory, but does not perform any of the operations specified by the program until you tell it to do so. You can enter as many program lines as you want. Then, when you enter the appropriate command, the computer performs the operations specified by the programmed mode program.

### Program Execution

We say the computer *executes* or *runs* a program when it performs the operations which the program specifies.

In immediate mode a program is executed as soon as you press the RETURN key.

In programmed mode you must issue the RUN command to execute a program. Each time you do so, the program runs all over again.

### Clearing Out Old Programs

Because the Apple II stores programmed mode programs in its memory, you must specifically instruct it to erase an old program before you type in a new program. Do this by typing the command NEW. If you forget to type NEW, your new program will be mixed in with your old program.

### Ending Programs Properly

The end of an immediate mode program is obvious. Not so with programmed mode, as we shall see. The END statement tells BASIC to stop executing your pro-

gram and return to immediate mode. Therefore an END statement should be the last statement your program executes.

Applesoft does not require an END statement. It ends a program automatically when it runs out of instructions.

## Line Numbers

*Line numbers* make programmed mode possible. A line number is simply a one, two, three, four, or five-digit number entered at the beginning of a program line. The line number is the only difference between a programmed mode program line and an immediate mode program line. There are some instructions that can be used only in immediate mode and others that can be used only in programmed mode; we will discuss them later.

Try this sample programmed mode program:

```
>NEW
>10 PRINT "RUBBER BABY BUGGY BUMPERS"
>20 END
>RUN
RUBBER BABY BUGGY BUMPERS
```

Each line number must be unique. No two program lines can have the same number. If you use the same line number more than once, the computer only remembers the last program line you used it with. To see how this works, type in these program lines:

```
>NEW
>10 PRINT "FIRST LINE 10"
>10 PRINT "SECOND LINE 10"
>20 END
>RUN
SECOND LINE 10
```

Line numbers determine the sequence of program lines in a BASIC program. The first line must have the smallest line number, while the last line must have the largest line number. Even if you type in the lines out of order, the Apple II will internally rearrange them in the proper sequence by line number. Consider this program, with line numbers out of order:

```
>NEW
>30 PRINT "CUT"
>10 PRINT "FISH"
>20 PRINT "OR"
>40 PRINT "BAIT"
>50 END
>RUN
FISH
OR
CUT
BAIT
```

To prove that the Apple II does not forget programmed mode in programs, clear the display screen with the Esc-@ command and then rerun the program.

> ← Press Esc-@, then RETURN

```
>RUN
FISH
OR
CUT
BAIT
```

It is a simple matter to add program lines to a program that is currently in the computer's memory. You can add a line to the beginning, the end, or anywhere in the middle of a program by typing the line with a line number that will position it where you want it. Suppose you wanted to add a line to the beginning of the last example program. As long as you have not typed the command NEW, the program will still be in the Apple II's memory. Since the lowest line number currently in that program is 10, any program line you type in now with a line number less than 10 will be placed at the beginning of the program. Try this:

```
>5 PRINT "EITHER"
>RUN
EITHER
FISH
OR
CUT
BAIT
```

It's a good thing the original program started with line 10 rather than line 0! It's always a good practice when assigning line numbers to start your program with a fairly high line number and leave plenty of room between line numbers so you can add program lines later on.

### Multiple-Statement Program Lines

You can put more than one statement on a single program line. The first statement follows the line number. The second statement follows the first, with a colon (:) in between. Colons separate the statements on a multiple-statement line.

Integer BASIC allows you to put more than one statement on a line in program mode, unlike immediate mode. Line length is limited to approximately 150 characters. The exact line length limit varies depending on the content of the line. If you type a line that is too long, the Apple II displays the error message \*\*\* TOO LONG ERR and you must retype the line.

Multiple-statement program lines are allowed in Applesoft in both programmed and immediate modes. In both cases, the line length limit is 255 characters, as we described earlier.

### Listing Program Lines

You can see what program lines the computer has stored in its memory at any time by typing the command LIST. Try it right now. If you have not typed NEW

since you tried the last example, you should see the following program lines displayed on the screen:

```
LIST
      5 PRINT "EITHER"
      10 PRINT "FISH"
      20 PRINT "OR"
      30 PRINT "CUT"
      40 PRINT "BAIT"
      50 END
```

This is called a *program listing*. There are variations of the LIST command which allow you to list one line at a time or a group of lines. This latter option is especially handy when you have a long program that will not fit on the display screen all at once. With the last example program still in the computer's memory, typing the command

```
LIST 10
```

causes the following program line to appear on the display screen:

```
10 PRINT "FISH"
```

To list several sequential program lines, you must specify both the starting and ending line number, as in this example:

```
LIST 20,40
      20 PRINT "OR"
      30 PRINT "CUT"
      40 PRINT "BAIT"
```

In Applesoft, you can list all program lines up to and including a specific program line. You can also list all program lines starting at a specific program line up to the end of the program. Here are examples of those two versions of the LIST command:

```
LIST,10

      5 PRINT "EITHER"
      10 PRINT "FISH"

LIST 30,

      30 PRINT "CUT"
      40 PRINT "BAIT"
      50 END
```

### Interrupting a Listing

You can halt a listing before it reaches the end by typing CTRL-C. This is especially useful for aborting the interminable listing of a long program.

If your Apple II has the Autostart Monitor, you can suspend, or temporarily freeze the listing of a program by typing CTRL-S. The listing will resume when you



press the space bar. CTRL-S allows you to review the listing of a long program at your own pace.

### Automatic Line Numbering

The Apple II will automatically number your Integer BASIC program lines for you. Use the AUTO command to institute this. The computer will then supply the next line number each time you press the RETURN key. It will not advance to the next line number if it finds an error on the line you just finished, or if you entered nothing on the line except RETURN.

To get out of the automatic line numbering mode, type CTRL-X. This cancels the line number provided by the computer. Following that, type the command MAN.

To see how the AUTO and MAN commands work, type in the following:

```
>AUTO 1000

>1000 PRINT "HOW MANY YARDS IN A MILE?"
>1010      Just press RETURN
>1010 PRINT 5280/3
*** SYNTAX ERR
>1010 PRINT 5280/3
>1020 \    Press CTRL-X
>MAN

>
```

As you can see from the example, AUTO requires you to specify which line number automatic line numbering should start with. You can also specify the increment between line numbers. The following example illustrates this option.

```
>AUTO 1000,100

>1000 PRINT "FISH OR CUT BAIT"
>1100 \
>MAN      Press CTRL-X
```

In this example, line numbers are incremented by 100. If you do not specify the increment, the Apple II defaults to 10 as in the previous example.

Applesoft does not have the automatic line numbering feature.

### SAVING PROGRAMS ON CASSETTE

A cassette recorder gives you the means of saving a programmed mode program outside the main computer and later loading that program back into memory. Suppose you have the following program in memory.

```
10 PRINT "TOTO,"
20 PRINT "I"
30 PRINT "DON'T"
```

```
40 PRINT "THINK"  
50 PRINT "WE'RE"  
60 PRINT "IN"  
70 PRINT "KANSAS"  
80 PRINT "ANYMORE"  
90 END
```

To save this program, put a tape in the cassette recorder. Rewind it to the beginning, then press the RECORD and PLAY buttons simultaneously on the cassette recorder and enter the following command right away at the keyboard:

```
SAVE
```

The Apple II will beep as it starts recording the program on the tape and will beep once again when it finishes. Press the STOP button on the recorder after the second beep.

At this point type in NEW followed by LIST to erase the program from the computer memory and verify that it is gone. To load the program back into the computer from the tape, first rewind the tape to the beginning. Then press the PLAY button on the cassette recorder and enter the following command at the keyboard:

```
LOAD
```

The Apple II beeps as it starts to load the program from the tape and beeps once again when it finishes. You must manually stop the tape after the second beep. Use the LIST command to verify that the program is in the computer memory.

In Chapter 5 we will see how to save and load programs onto disk, which is even more convenient than cassette tape.

### **Saving Multiple Programs on One Tape**

You may have noticed that it did not take very much tape to save the one program. A longer program would require more tape, but usually there is enough tape on one cassette to hold several BASIC programs. You can save programs sequentially on the tape: the second follows the first, the third follows the second, and so on.

Loading the second, third, and subsequent programs on a cassette is not as straightforward as loading the first. After you rewind the tape to the beginning, you must get past the first program in order to load the second, past the second to get at the third, etc. You can do this by typing the LOAD command repeatedly until the program you want is in memory. This is a slow process, but it works.

You can speed things up considerably if your cassette recorder has a tape counter. Reset the tape counter to 0 when you rewind the tape to the beginning before saving a program. After saving the first program, jot down the tape counter reading. This is the starting tape counter reading for the second program. Save the second program and note the tape counter reading at the end of it (for the start of the third program).

Now to load the second program, rewind the tape to the beginning and reset

the tape counter to 0. Then use the FAST FORWARD button on the cassette recorder to position the tape counter to the reading for the start of the second program. You can use the REWIND button on the cassette recorder to back the tape up if you overshoot with the FAST FORWARD button. Now use the LOAD command to get the second program.

## SWITCHING BASICS

On many versions of the Apple II, you have a choice between programming in Integer BASIC and Applesoft. The reasons for choosing one version of BASIC over the other will become apparent as you progress through this chapter. We have already seen, for example, that Integer BASIC will provide program line numbers automatically, while Applesoft will not. On the other hand, Applesoft allows you to use numeric values with fractions, while Integer BASIC does not.

While most versions of the Apple II have both versions of BASIC available, not all do. The standard Apple II Plus has only Applesoft, for instance. The procedure for switching from one version of BASIC to the other varies depending on which model of Apple II you have, and which options are installed on it.

With the Apple Language System option, you have immediate access to either version of BASIC. If you are in Integer BASIC, type FP to transfer to Applesoft. From Applesoft, type INT for Integer BASIC.

The Applesoft firmware card also allows you immediate access to either version of BASIC. It has a switch which protrudes through a slot on the back of the Apple II and determines the version of BASIC. You will get Integer BASIC if you turn on the Apple II with the firmware card switch in the down position. When you turn the Apple II on with the firmware card switch in the up position you get Applesoft. Regardless of the switch setting, you can type FP for Applesoft or INT for Integer BASIC if the firmware card is installed.

On a standard Apple II it is easy to get Integer BASIC. Press the RESET key, and then type CTRL-B. It is harder to get Applesoft on a standard Apple II since it resides on either cassette or disk. You must instruct the Apple II to load Applesoft into its memory from the disk or cassette.

Before you can use the disk drive you must *boot* the Disk Operating System (DOS) as described in Chapter 2. You must do this each time you turn the computer back on after it has been off. To review briefly, here are the standard instructions for booting DOS from the Monitor and Integer BASIC:

\*6. Press CTRL-P, then RETURN

>PR#6

Once DOS is booted, you can put the Apple II in Applesoft from Integer BASIC by placing a disk with Applesoft on it in the disk drive and then typing FP. In a few

seconds the Applesoft prompt character (>) will appear on the display screen.

You can also get Applesoft from cassette tape. The complete procedure for doing this is described in Chapter 2. Briefly, you must put the Applesoft cassette in the cassette recorder, press the PLAY button on the cassette recorder, and type the LOAD command from Integer BASIC. In 1-1/2 or 2 minutes, the Applesoft copyright notice and prompt character will appear on the display screen.

From Applesoft, you can switch back to Integer BASIC by typing the command INT. If you then wish to switch back to Applesoft, use the FP command with the disk drive, or the LOAD command with the cassette tape.

## ADVANCED EDITING TECHNIQUES

In Chapter 2 we looked at ways you could correct errors in a line you are typing before you press the RETURN key. Let's quickly review those simple editing techniques.

The ← key backspaces the cursor and erases characters it passes over.

Characters are erased from the program line even though they still appear on the display screen.

The → key moves the cursor forward, copying over (retyping) characters it passes over.

The REPT key, used in conjunction with the ← and → keys, enables fast-backspacing and fast-forwarding.

The CTRL-X command cancels the line you're currently typing.

The Esc-@ command clears the display screen and leaves the cursor in the upper left corner.

We will now discover new ways to edit program lines. These new methods are particularly useful when you want to make changes to programmed mode lines (i.e., those with line numbers).

## DELETING PROGRAM LINES

To delete an entire line, type its line number followed immediately by a RETURN. When you list the program, you will see that the line and line number are no longer part of the program. Here is an example:

```
>NEW

>100 PRINT "VIRTUE IS ITS OWN REWARD"
>110 PRINT "IF THE SHOE FITS, WEAR IT"
>120 PRINT "WHERE THERE'S SMOKE, THERE'S
FIRE"
>130 PRINT "LOOK BEFORE YOU LEAP"
>140 PRINT "BREVITY IS THE SOUL OF WIT"
```

```

>150 END
>110
>130
>LIST
  100 PRINT "VIRTUE IS ITS OWN REWARD"

  120 PRINT "WHERE THERE'S SMOKE, THERE'S FIRE"
  140 PRINT "BREVITY IS THE SOUL OF WIT"
  150 END

```

You can use the command DEL to delete a block of program lines. For example:

```

>DEL 110,140

>LIST
  100 PRINT "VIRTUE IS ITS OWN REWARD"

  150 END

```

The command DEL 110,140 deletes all program lines starting at line number 110 and ending with line number 140. Even though line 110 does not exist, all lines between 110 and 140 are deleted.

### ADDING PROGRAM LINES

You can type in new program lines in any order, at any time. Their line numbers will determine their position in the program. The Apple II will automatically merge them with any other program lines currently in memory. Try adding lines 120 and 110 back into the example above.

```

>120 PRINT "WHERE THERE'S SMOKE, THERE'S FIRE"
>110 PRINT "IF THE SHOE FITS, WEAR IT"
>LIST
  100 PRINT "VIRTUE IS ITS OWN REWARD"

  110 PRINT "IF THE SHOE FITS, WEAR IT"
  120 PRINT "WHERE THERE'S SMOKE, THERE'S FIRE"
  150 END

```

### CHANGING PROGRAM LINES

The simplest way to change a program line is to retype it. This is unsatisfactory for several reasons. Retyping is a time-consuming chore and the chances of typographical errors are high. Fortunately there is a way to modify program lines you have already typed into the computer memory. The feature of both Integer BASIC and Applesoft that makes this possible is that anything displayed on the

screen is *live*. You can edit anything on the screen. By using the Esc key in conjunction with several other keys, you can move the cursor around on the screen at will. This allows you to position the cursor to the beginning of any line that is displayed on the screen. Then you can use the → key to copy over unchanged parts of the program line. You can replace, insert, or delete characters anywhere on the line.

Here is how it works. First, you use the LIST command to display the program line or lines you wish to change. You may have noticed that the LIST command puts in extra spaces when it displays program lines. (This is to make the program more readable.) These extra spaces can make it more difficult to edit longer program lines. To keep the LIST command from adding extra spaces when it displays program lines, clear the display screen (use Esc-@) and type the following mystery command:

```
POKE 33,33
```

In addition to suppressing the extra spaces, this command reduces the width of the display screen from 40 characters to 33 characters. We will cover the POKE statement in more detail in Chapter 4. To get the display back to normal, type:

```
POKE 33,40
```

### Moving the Cursor

To move the cursor around on the screen, you must press two keys in sequence. First press the Esc key and then press either the A, B, C, or D key. You must press Esc followed by A, B, C, or D each time you want to move the cursor one position right, left, down, or up. Figure 3-1 illustrates how the four possible key sequences affect cursor movement.

With the Autostart Monitor, you can also use the I, J, K, and M keys in conjunction with the Esc key to move the cursor. Because of the way these four keys are situated on the keyboard, they form a directional control pad as shown in Figure 3-2.

There is an important difference in the way Esc works with I, J, K, and M. Press Esc and the Apple II goes into *edit mode*. Now press I, J, K, and M to move the

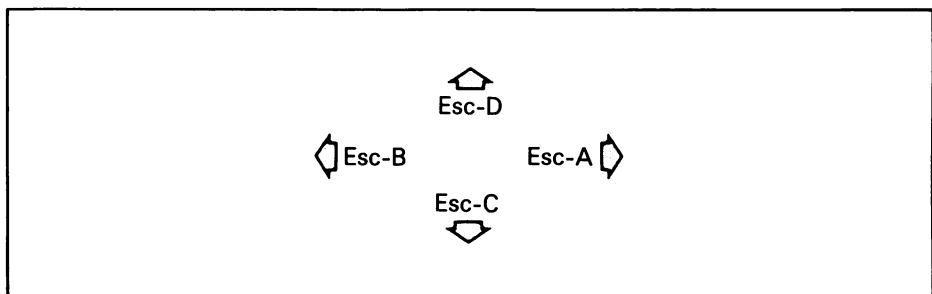


FIGURE 3-1. Cursor Movement (Two-Key Sequences)

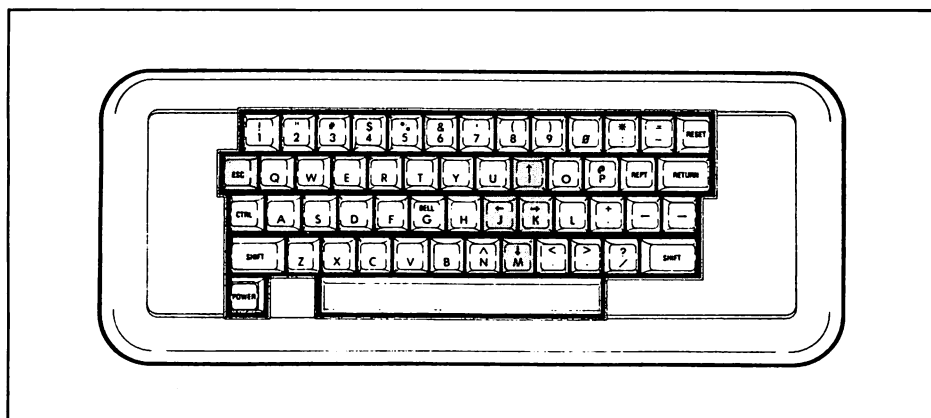


FIGURE 3-2. Cursor Movement (Autostart Monitor Version)

cursor around on the display screen — no need to press Esc each time. The Apple II stays in edit mode as long as you use the I, J, K, M, or REPT keys. This allows you to move the cursor more than one space without having to retype Esc each time. To get out of edit mode, press any key except the I, J, K, M, REPT, CTRL, or SHIFT keys.

You can use the REPT key in edit mode to move the cursor longer distances with fewer keystrokes.

### Changing Characters

Replacing one character with another is simplicity itself. Merely position the cursor on the offending character and type the replacement right over it. For example, with the cursor as shown:

```
100 PRINT "ESTIMATED TIME OF ARRIVAL"
```

you can type the word DEPARTURE and get this:

```
100 PRINT "ESTIMATED TIME OF DEPARTURE"
```

Press RETURN to effect the change.

### Deleting Characters

You can effectively delete individual characters by typing over them with blank spaces. Remember that in BASIC extra blank spaces do not affect anything unless they are inside of quotation marks. You can also use the Esc and A keys (the K key in edit mode) to move the cursor forward. Unlike the → key, Esc-A and Esc-K do not recopy characters they pass over. If the characters you want to delete are inside quotation marks, it is easier to use the Esc and A keys, or the J key in edit mode, to skip over unwanted characters.

To blank out all characters from the cursor position to the end of the *display* line, press Esc and then E. This has the same effect as pressing the space bar repeatedly until you reach the end of the display line, except the cursor doesn't move. Characters on the next display line are not erased from the screen even if they were part of the same program line. For example, if you press Esc and then E with the cursor positioned like this:

```
100 PRINT "IT IS BETTER TO HAVE LOVED AN
D LOST THAN NEVER TO HAVE LOVED AT ALL"
```

this is what happens:

```
100 PRINT "IT IS BETTER TO HAVE
D LOST THAN NEVER TO HAVE LOVED AT ALL"
```

Beware! If you press RETURN right now, line 100 will end right where the cursor is. Use the → key to recopy the rest of the program line.

You can also clear all the text from the cursor position to the end of the display screen. To do this, press Esc and then F.

### Inserting Characters

Inserting characters into a line is an easy process. It may seem confusing at first because the final results are not immediately apparent. The Apple II cannot push apart characters on a line to make room for insertions. Figure 3-3 diagrams the way the Apple II lets you perform insertions. You insert text above the line with the aid of the cursor movement keys (Esc, etc.). The thing you must remember is that what is displayed on the screen is not an exact replica of what is stored in the computer memory.

Here is a sample editing session that demonstrates character insertion. The sample uses the Integer BASIC prompt character (>) but will work exactly the same in Applesoft. Consider the following program line:

```
>NEW
>10 PRINT "ON THE WAGON"
>
```

To insert the word BAND in front of the word WAGON, first list the program:

```
>LIST
10 PRINT "ON THE WAGON"
>
```

Using the cursor movement keys (Esc, etc.) *only*, position the cursor so that it is directly over the first digit of the line number, as follows:

```
>LIST
10 PRINT "ON THE WAGON"
>
```

Now use the → key to copy over the first part of the line. Stop at the W.



BAND 10 PRINT "ON THE WAGON"
---------------------------------

FIGURE 3-3. Character Insertion

```
>LIST
  10 PRINT "ON THE WAGON"
>
```

Press the Esc key followed by the D key to move the cursor up one line. (If there are characters to the right of the cursor on this line, you can erase them by pressing the Esc key followed by the E key.)

```
>LIST
  10 PRINT "ON THE WAGON"
>
```

Type the word BAND.

```
>LIST
  10 PRINT "ON THE WAGON"
>
```

Using *only* the cursor movement keys, position the cursor over the first letter of the insertion. Do *not* use the ← key to back the cursor up. Although it looks like it does the same thing on the display screen as Esc-B or Esc-J, ← actually erases the characters it passes over; it will un-insert your insertions!

```
>LIST
  10 PRINT "ON THE WAGON"
>
```

Press the Esc key followed by the C key to return the cursor to the original program line.

```
>LIST
  10 PRINT "ON THE WAGON"
>
```

Finally, use the → key to copy over the rest of the line. Then press RETURN. Display the new line with the LIST command.

```
>LIST
  10 PRINT "ON THE WAGON"

>LIST 10
  10 PRINT "ON THE BANDWAGON"
>
```

Appendix B contains a handy reference table of editing commands.

## REEXECUTING IN IMMEDIATE MODE

The fact that anything on the Apple II display screen is live allows you to reexecute any immediate mode statements that are still visible on the display screen. You can reexecute an immediate mode statement just as it is, or you can edit it first.

In either case, the first thing to do is to position the cursor to the beginning of the immediate mode line. Use the Esc and A, B, C, D keys alternately as described above (or Esc followed by any number of I, J, K, and M keys if you have an Autostart Monitor). Then use the → key to copy over the immediate mode instruction. You can, of course, make changes to the line using the techniques we have just described for replacing, deleting, and inserting characters on a line.

To see how this works, look at the following immediate mode program which calculates the cubic feet of storage space in a  $10 \times 25 \times 8$  foot room.

```
>PRINT "CU. FT. OF SPACE = ";10*25*8  
CU. FT. OF SPACE = 2000
```

You can change this immediate mode program easily to calculate the storage space in rooms of different sizes. To change the dimensions to  $10 \times 25 \times 14$ , for example, first position the cursor to the beginning of the immediate mode line. (Alternately press Esc and D three times.) Now press and hold the → key while you hold down the REPT key. The cursor will fast-forward along the immediate mode line. Release both keys in time to stop the cursor when it gets to the 8. If you overshoot or undershoot by not releasing the keys at the proper time, you can move the cursor back and forth one character at a time with the ← and → keys. For that matter, you could move the cursor from the start of the line to the 8 by pressing the → key 33 times, instead of using the REPT and → keys in conjunction.

With the cursor positioned over the 8, type in the new room dimension of 14 and press RETURN.

```
>PRINT "CU. FT. OF SPACE = ";10*25*14  
CU. FT. OF SPACE = 3500
```

## PROGRAMMING LANGUAGES

A programming language is the medium of communication between you and the computer. There are many different programming languages. Some, like BASIC, are general purpose languages, while others are designed to make it easy to write programs in specific areas like business, science, graphics, text manipulation, and so forth. Programming languages are as different as spoken languages. In addition to BASIC, other common programming languages are Pascal, C, FORTRAN, COBOL, APL, PL/M, PL-1, and FORTH.

Apple II computers can use several programming languages, BASIC and Pascal among them. This book concentrates on describing how to program the Apple II in BASIC.

No matter what the programming language, every program statement must be written following a well-defined set of rules. These rules taken together are referred to as *syntax*. Each different programming language has its own syntax.

Programming languages, like spoken languages, have *dialects*. Dialects manifest themselves as minor variations in syntax. The Apple II has two such dialects of BASIC: Integer BASIC and Applesoft. Because of the differences between the two dialects, very often programs written in one dialect will not work correctly when the Apple II is expecting instructions in the other dialect. Furthermore, a BASIC program written for the Apple II may not run on another computer, even if the other computer also claims to be programmable in BASIC. However, having learned how to program your Apple II computer in either of its BASIC dialects, you will have little trouble learning any other dialect of BASIC.

Some programming language syntax rules are obvious. The addition and subtraction examples at the beginning of this chapter use obvious syntax. You do not have to be a programmer to understand them. But most syntax rules are utterly arbitrary and meaningless, until you have learned the syntax. You should not try to seek a rationale for syntax rules; usually there is none. For example, why use \* to represent multiplication? Normally, you would use an X for multiplication. But the computer would have no way of differentiating between the use of X to represent multiplication and to represent the letter X. Therefore, nearly all computer languages have opted for the asterisk (\*) to represent multiplication. Division is universally represented by the / sign. There is no real reason for this selection; the division sign ( $\div$ ) is not present on computer keyboards, so some other character must be selected.

## ELEMENTS OF BASIC

Most of the syntax rules for BASIC concern individual statements. BASIC statement syntax deals separately with its three major elements: line numbers, instructions to the computer, and data. We will describe each in turn. There are also a few rules that pertain to the program as a whole, such as statement order. We will cover these rules in appropriate places throughout the chapter.

### LINE NUMBERS REVISITED

We've already talked about line numbers to some extent. After a brief review, we'll go into more detail. In programmed mode, every line of a BASIC program must have a unique line number. Line numbers determine the sequence of instructions in a program; the statement with the lowest line number is first and the statement with the highest line number is last.

Integer BASIC allows one- to five-digit line numbers with integer values between 0 and 32767.

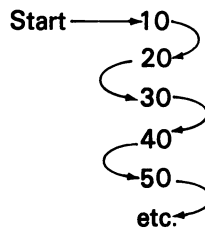
Applesoft allows one- to five-digit line numbers with integer values between 0 and 63999.

### Line Numbers as Addresses

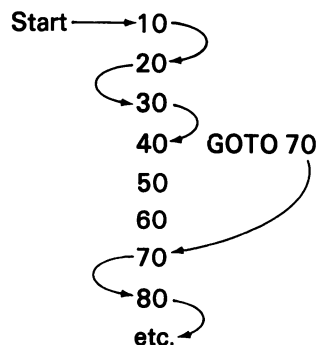
In essence, line numbers are a way of addressing program lines. This is an important concept, since every program will contain two types of statements:

1. Statements which create or modify data, and
2. Statements which control the sequence in which operations are performed.

The idea that operations specified by a program must be performed in some well-defined sequence is a simple enough concept. Normally, program execution begins with the first statement in the program and continues sequentially, as illustrated below.



But we will soon discover that most programs contain some non-sequential execution sequences. That is when line numbers become important, because you can use a line number to identify a change in execution sequence. This may be illustrated as follows:



### BLANK SPACES

Generally speaking, you can use extra blank spaces freely to improve the readability of your program. But because each extra space would take extra memory, the Apple II compresses unnecessary blank spaces out of a program line

when you enter it (and press RETURN). Then when you display program lines with the LIST command, the Apple II reinflates them with blank spaces for readability, according to a predetermined plan. Recall that you can suppress the reinflation by typing POKE 33,33 before issuing a LIST command. POKE 33,40 normalizes the display screen.

You do have to be careful about using extra spaces within quotation marks. Compare the following two commands, for example:

```
PRINT "ENTER INVOICE DATE"
ENTER INVOICE DATE

PR INT "      ENTER INVOICE D ATE"
      ENTER INVOICE D ATE
```

## DATA

The main business of computer programs is to input, manipulate, and output data. So the way a programming language handles data, whether it be numbers or text, is very important indeed. We will now describe the types of data you may encounter in an Apple II BASIC program.

### Strings

A *string* is any character or sequence of characters enclosed in quotation marks. We have already used strings with the PRINT statement as messages to be displayed on the screen. Here are some more examples of strings:

```
"IGNORANCE IS BLISS"
"ACCOUNT 4019-181-324-837"
"NICK CHARLES"
"SAM & ELLA CAFE"
"MARCH 18, 1956"
```

With just a few exceptions, a string can contain any character you can produce at the keyboard using the normal alphabetic and numeric keys, with or without the CTRL or SHIFT keys. The exceptions are ←, →, RETURN, Esc, CTRL-H, CTRL-M, CTRL-U, and CTRL-X. These exceptions either move the cursor around or end the line you're working on, or both.

Strings can be any length from 0 to 255 characters. A string with no characters in it is called the *null string*.

There are some invisible characters you can produce by pressing certain combinations of keys. For example, if you press the CTRL and G keys simultaneously, the computer beeps. You can put this character in a string. Try using some of these characters with a PRINT statement.

```
PRINT " "  Press CTRL-G several times between
           the quotation marks
```





As you can see, scientific notation is a convenient way of expressing very large and very small numbers. The maximum and minimum values for real numbers, which we just expressed with lots of zeros, can also be expressed as  $1\text{E}+38$  and  $-1\text{E}+38$ , respectively (much more compact). Similarly, the closest a number can get to zero is  $3\text{E}-38$ .

### Roundoff

We mentioned earlier in this chapter that real numbers can have as many as nine digits of precision. For a number greater than 1 or less than  $-1$ , this means only the leftmost nine digits can be nonzero. The Apple II rounds off any digits in excess of nine. Here are some examples (note that large numbers print in scientific notation):

```
JPRINT 1234567891
1.23456789E+09
```

```
J?-123456789123456789
-1.23456789E+17
```

```
J?-150000475.75
-150000476
```

```
J?900000000.7558
900000000.8
```

Fractional numbers (those between 1 and  $-1$ ) are subject to the same limitation. In this case, though, the nine digits of precision start with the first nonzero digit to the right of the decimal point. Here are some examples:

```
JPRINT .1234567891
.123456789
```

```
J?-123456789123456789
-1.23456789E+17
```

```
J?-123456789 123456789
-1.23456789E+17
```

```
J?.000000000900000007558
9.00000008E-10
```

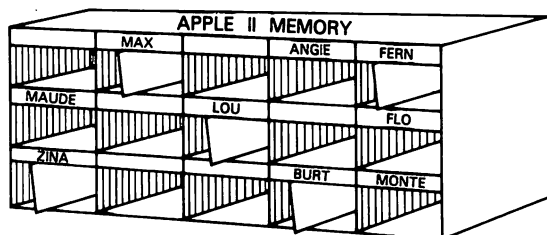
### VARIABLES

Thus far in our discussions of data we have only considered constant values. It is often handier to refer to data items by name rather than value. That is what *variables* are all about.

If you have studied elementary algebra, you will have no trouble understanding the concept of variables and variable names. If you have never studied algebra, then think of a variable name as a name which is assigned to a letter box. Anything which is placed in the letter box becomes the value associated with the letter box name, until something new is placed in the letter box. In computer jargon



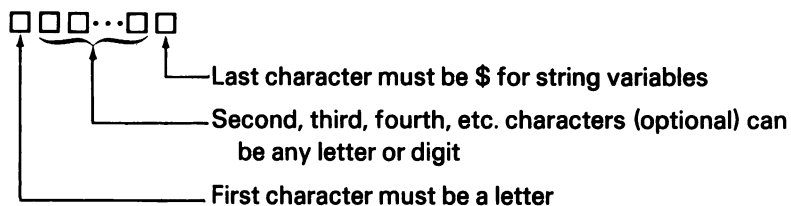
we say a value is stored in a variable.



A variable does not always have to refer to the same value. That is its real power — it can represent any legal value. You can change its value during the course of a program. BASIC has a number of statements to do this; we will describe them later.

### Integer BASIC Variable Names

Variable names in Integer BASIC can have 1 to 100 characters. These are the general rules for constructing Integer BASIC variable names:



Thus the last character of the variable name tells Integer BASIC which type of data the variable represents — string or numeric.

*String variables* can refer to strings of any length between 0 and 255 characters. Blank spaces in a string count towards its total length. Before you use a string variable, you must specify the maximum length it will have. You do this with the DIM statement, which we will describe later. If you fail to do so, you will get a \*\*\* STR OVFL ERR message. Here are some examples of string variable names, legal and illegal:

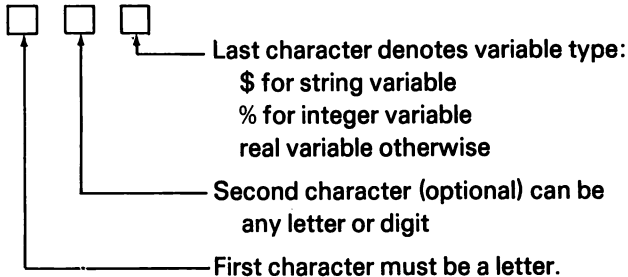
Legal	Illegal
A\$	\$
CUSTNAME\$	9\$
PART1\$	BRAND.NAMES\$
RESPONSE\$	
X8\$	

*Numeric variables* in Integer BASIC must have values between  $-32767$  and  $+32767$ . If you surpass these bounds, you will get the `*** > 32767 ERR` message. Here are some examples of numeric variable names in Integer BASIC, both legal and illegal:

Legal	Illegal
A	APPLICANT'S AGE
CUSTZIPCODE	3X4Z
X0	\$TOTAL

### Variable Names in Applesoft

A variable name can have one, two, or three characters in Applesoft. The following rules apply:



Thus the last character of the variable name tells Applesoft which type of data the variable represents.

A *string variable* in Applesoft can store a string value of any length from 0 to 255 characters. You do not specify a maximum string length as in Integer BASIC; Applesoft does not need it. Here are some examples of string variable names, both legal and illegal:

Legal	Illegal
A\$	0\$
MN\$	MI\$
F6\$	77\$

*Integer variables* can refer to whole numbers between  $-32767$  and  $+32767$ . If you attempt to exceed this limit, you will get the `?ILLEGAL QUANTITY ERROR` message. If you try to store a real value in an integer variable, Applesoft will convert the real value to an integer value first. We'll cover the rules for the conversion shortly. Here are some examples of legal and illegal integer variables in Applesoft:

Legal	Illegal
A%	A\$%
B%	31%
A1%	3D%
X4%	

*Real variables* can refer to numeric values generally restricted to the range  $-10^{38}$  to  $+10^{38}$  although you may be able to compute values as large as  $1.7 \times 10^{38}$  in magnitude (i.e., + or -) under some circumstances. If you attempt to store (in a real variable) a value that is too large in magnitude you will get the ?OVERFLOW ERROR message.

When the value of a real variable gets closer to zero than  $\pm 2.9388 \times 10^{-39}$ , Applesoft converts it to zero.

Remember, real variables can have integer values, since an integer is a real number with a fractional part of zero. Here are some examples of real variables (legal and illegal):

Legal	Illegal
A	0
B	7B
A1	A#
AA	
Z5	

### Longer Variable Names in Applesoft

Variable names can actually have more than two alphanumeric characters (plus the % or \$ suffix for integer- and string-type variables), but only the first two characters count in Applesoft. Therefore, PRICE1 and PRICE2 are the same name, since both begin with PR. However, PRICE1 and PRICE1% are different, since they have different type suffixes.

Applesoft allows variable names to have up to 238 characters.

Here are some examples of variable names with more than two characters:

Legal	Illegal
COUNTER%	ITEM#%
ACCOUNTBALANCE	2NDRATE
NAME\$	CUSTOMER. ADDRESS\$

Keep the following points in mind if you use variable names with more than two characters:

1. Only the first two characters and variable type suffix (\$ or %) are significant. Do not use extended names like LOOP1% and LOOP2%; these refer to the same variable: LO%.
2. Additional characters need extra memory space, which you might need for longer programs. But the advantage of using longer variable names is that they make programs easier to read. PARTNO, for example, is more meaningful than PA as a variable name describing part numbers in an inventory program.

### Reserved Words

All of the words that define a BASIC statement's operations are called *reserved words*. Appendix F lists all Integer BASIC and Applesoft reserved words. You will

have encountered many of these reserved words in this chapter, but others are described elsewhere in this book.

When executing BASIC programs, the Apple II scans every BASIC statement, seeking out any character string that constitutes a reserved word. The only exception is text strings enclosed in quotes. This can cause trouble if a reserved word is embedded anywhere within a variable name. The Apple II is not smart enough to identify a variable name by its location in a BASIC statement. *Therefore you must be very careful to keep reserved words out of your variable names;* this is particularly important with the short reserved words that can easily slip into a variable name.

## ARRAYS

*Arrays* are really nothing more than a systematic way of naming a large number of variables. They are used frequently in many types of computer programs. If you do not understand what arrays are, or how to use them, then read on; the information that follows will be very important to your programming efforts.

Conceptually, arrays are very simple things. When you have two or more data items, instead of giving each data item a separate variable name, you give the collection of data items a single variable name. The collection is called an *array*, its name is an *array name*. Individual data items are often called *array elements*. The elements in an array are numbered. You select an individual item using its position number, which is referred to as its *index*.

Arrays are a useful shorthand means of describing a large number of related variables. Consider, for example, a table of 200 numbers. How would you like it if you had to assign a unique variable name to each of the 200 numbers? It would be far simpler to give the entire table one name, and identify individual numbers within the table by their table location. That is precisely what an array does.

As an example of array usage, consider how a motel with ten rooms might keep track of who is staying in each of its rooms. There could be a separate variable name for each room.

JONES	SMITH	DOE		LITKE	ALTON	DAVIS	HANSON	SHORTEN	
R1\$	R2\$	R3\$	R4\$	R5\$	R6\$	R7\$	R8\$	R9\$	R10\$

Or you could consider the motel guest list an array.

JONES	SMITH	DOE		LITKE	ALTON	DAVIS	HANSON	SHORTEN	
R\$(1)	R\$(2)	R\$(3)	R\$(4)	R\$(5)	R\$(6)	R\$(7)	R\$(8)	R\$(9)	R\$(10)

In this example, R\$ is an array name. It has ten elements; each element is the name of the guest in one of the rooms. An index (enclosed in parentheses) follows each variable name. Thus a specific data item (i.e., the guest in one room) is identified by a variable name and an index. The guest in the third room is the value of R\$(3), which is DOE.

The array examples above all use string arrays because strings are easier for most people to understand intuitively. Actually, Integer BASIC only allows numeric arrays.

Arrays in Applesoft can represent integer variables, real variables, or string variables; however, a single array variable can only represent one data type. In other words, a single variable cannot mix integer and real numbers, except in the sense that a real variable can have an integer value (but not vice versa). Each type of array uses a different amount of memory; see Appendix G for details.

### Array Dimension(s)

You must specify the number of elements in an array before you use it in Integer BASIC. You do this with a DIM (*dimension*) statement. Dimension statements are described later in this chapter.

Applesoft lets you use arrays with up to ten elements without dimensioning them first.

Applesoft arrays can have more than one dimension, which means that it will take more than one index to select an individual element. An array with a single dimension is equivalent to a table with just one row of numbers. The index identifies a number within the single row. An array with two dimensions yields an ordinary table of numbers with rows and columns; one index identifies the row, the other index identifies the column. You can visualize an array with three dimensions as a cube of numbers, or perhaps a stack of tables. Four or more dimensions yield an array that is hard to visualize, but mathematically no more complex than a smaller dimensioned array.

We can create an example of a two-dimensional array by extending the previous example of the motel guest list. Consider an eight-floor hotel with ten rooms on each floor. There are four options for keeping track of the 80 guests' names. First, each room could have its own variable. Second, the hotel could have one 80-element array. Third, each floor could have a separate ten-element array. Fourth, the hotel could have one two-dimensional array. This final choice may be illustrated as follows:

H\$(8,1)	H\$(8,2)	H\$(8,3)	H\$(8,4)	H\$(8,5)	H\$(8,6)	H\$(8,7)	H\$(8,8)	H\$(8,9)	H\$(8,10)
H\$(7,1)	H\$(7,2)	H\$(7,3)	H\$(7,4)	H\$(7,5)	H\$(7,6)	H\$(7,7)	H\$(7,8)	H\$(7,9)	H\$(7,10)
H\$(6,1)	H\$(6,2)	H\$(6,3)	H\$(6,4)	H\$(6,5)	H\$(6,6)	H\$(6,7)	H\$(6,8)	H\$(6,9)	H\$(6,10)
H\$(5,1)	H\$(5,2)	H\$(5,3)	H\$(5,4)	H\$(5,5)	H\$(5,6)	H\$(5,7)	H\$(5,8)	H\$(5,9)	H\$(5,10)
H\$(4,1)	H\$(4,2)	H\$(4,3)	H\$(4,4)	H\$(4,5)	H\$(4,6)	H\$(4,7)	H\$(4,8)	H\$(4,9)	H\$(4,10)
H\$(3,1)	H\$(3,2)	H\$(3,3)	H\$(3,4)	H\$(3,5)	H\$(3,6)	H\$(3,7)	H\$(3,8)	H\$(3,9)	H\$(3,10)
H\$(2,1)	H\$(2,2)	H\$(2,3)	H\$(2,4)	H\$(2,5)	H\$(2,6)	H\$(2,7)	H\$(2,8)	H\$(2,9)	H\$(2,10)
H\$(1,1)	H\$(1,2)	H\$(1,3)	H\$(1,4)	H\$(1,5)	H\$(1,6)	H\$(1,7)	H\$(1,8)	H\$(1,9)	H\$(1,10)

As you can see, the first index of this two-dimensional array is the floor number, and the second index is the room number on that floor. So R\$(3,2) would be the name of the guest in the second room on the third floor.

Applesoft arrays can have up to 88 dimensions. There is no specific limit on the number of elements in each dimension. The amount of memory available will limit the total number of elements, of course, since each element requires a certain amount of memory space.

## EXPRESSIONS

In the following section we will explore ways in which you can combine the values of variables and constants by using *expressions*. We have already used expressions to calculate the value of simple arithmetic problems in immediate mode. Recall that the statement

```
PRINT 4+6
10
```

tells the Apple II computer to add 4 and 6, and then display the sum. The statement

```
PRINT A+B
0
```

tells the Apple II computer to add the values of the two numeric variables A and B, and then display the sum.

The plus sign (+) specifies addition. Standard computer jargon refers to the plus sign as an *operator*. The plus sign is an *arithmetic operator* because it specifies addition, which is an arithmetic operation.

Arithmetic operators are easy enough to understand; we all learn to add, subtract, multiply and divide in early childhood. But there are other types of operators: *string operators*, *relational operators*, and *Boolean operators*. These are also easy to understand, but they take a little more explanation since they involve more abstract notions.

Each category of operators defines a type of expression. There are arithmetic expressions, string expressions, relational expressions, and Boolean expressions.

## Precedence of Operators in Expressions

Expressions can call for more than one operation to occur. For example, this statement:

```
PRINT A+B/10
0
```

calls for both addition and division in the same expression. There is a standard scheme for determining in what order to evaluate an expression. We will lay out these rules of *precedence* for each type of expression starting with string concatenation, then integer, real, relational, Boolean, and mixed-type expressions, in that order. First let's look at a way to override the standard precedence rules.

### Overriding Standard Precedence

You can change the order in which the Apple II evaluates expressions through the use of parentheses. Any operation within parentheses is performed first. When more than one set of parentheses is present, the Apple II evaluates them from left to right.

When one set of parentheses is enclosed within another set, it is called *nesting*. In this case, the Apple II evaluates the innermost set first, then the next innermost, etc. Parentheses can be nested to any level. You may use them freely to clarify the order of operations being performed in an expression.

Here are some examples of the immediate mode arithmetic calculations using parentheses:

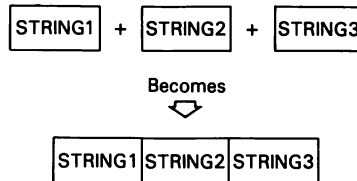
```
PRINT (2+10)*3
36

>
PRINT ((2+10)*3+31)*10
670

>
PRINT -(2^(3+8/4))
-32
```

### String Concatenation

You can join strings together end to end, to form one longer string. This is called *concatenation*. You can visualize this as follows:



With concatenation, you can develop strings up to 255 characters long.

Integer BASIC has no concatenation operator. You can concatenate strings in Integer BASIC by using a technique explained at the end of this chapter (introducing it here would be premature).

Applesoft uses the plus sign (+) as a concatenation operator. Here are some examples of string concatenation in Applesoft.

"OVER" + "DUE"	becomes	"OVERDUE"
"MONTHLY" + " " + "REPORT"	becomes	"MONTHLY REPORT"
"WEEKLY" + R\$	becomes	the characters WEEKLY followed by the value of R\$
A1\$ + YA\$ + C\$(1)	becomes	the value of A1\$ followed by the value of YA\$ followed by the value of C\$(1)

## Integer Expressions

Integer expressions are arithmetic expressions which involve only integer variables and integer constants. We will cover arithmetic expressions involving both integer and real values under the heading "Mixed-Type Expressions."

The operators for integer expression are addition (+), subtraction (-), multiplication (\*), division (/), and exponentiation (^). You can also use a unary minus (-) to indicate a negative numeric value. Operations are performed in this order: unary minus first, followed by exponentiation, multiplication and division next, and finally addition and subtraction. Operations of equal precedence are performed in order from left to right.

Here are some examples of integer expressions in Integer BASIC:

$100 - 30 * 2$	results in	40
$-9^2$	results in	81
$A/B*C$	results in	the value of A divided by the value of B and the integer value of the quotient multiplied by the value of C
$D + X*3$	results in	three times the value of X and the value of D added to that product
$5/2*2$	results in	4 (the quotient of 5/2 is converted to the integer 2)

Here are some examples of integer expressions in Applesoft:

$-120/2 + 100$	results in	40
$2^3*2$	results in	16
$N1%*N2\%/N3\%$	results in	the value of N1% times the value of N2% and the product divided by N3%
$AA\%/AB\%/AC\%$	results in	the value of AA% divided by the value of AB% and the quotient divided by the value of AC%
$5/2*2$	results in	5 (the quotient of 5/2 is not converted to an integer)

Integer BASIC has one more operator you can use in integer expressions. It returns the remainder that is left over from a division operation where the dividend is not evenly divisible by the divisor. The operator is MOD. It has equal precedence with multiplication and division. Here are some examples of MOD:

$4 \text{ MOD } 3$	results in	1
$3*5 \text{ MOD } 4$	results in	3
$(41+2)/25 \text{ MOD } A$	results in	the remainder after dividing 18 by the value of A
$3 \text{ MOD } 4$	results in	3



**Real Expressions**

Applesoft has another kind of arithmetic expression; it yields a real value. Its operators are the same as those in Applesoft integer expressions: addition (+), subtraction (−), multiplication (\*), division (/), exponentiation (^) and unary minus (−). The precedence of operation is the same also: unary minus first, followed by exponentiation, multiplication and division, and finally addition and subtraction. Here are some examples of real expressions:

87.5 − 4.25*2	results in	79
1.5 ^ (3/2/2)	results in	1.35540301
AL*(PL −3.1*CB)	results in	the value of AL times the result of subtracting the product of 3.1 times the value of CB from the value of PL
7.5*2/5	results in	3

**Relational Expressions**

Relational operators allow you to compare two values to see what relationship one bears to the other. You can compare whether the first is greater than, less than, equal, not equal, greater than or equal, or less than or equal to the second value. The values you compare can be constants, variables, or any kind of expressions. (There are some restrictions in Integer BASIC.) If the value on one side of a relational operator is a string, the value on the other side must be a string also. Otherwise, you can compare one type of value to another type using relational operators.

If the relationship is true, the relational expression has a numeric value of 1. If the relationship is false, the relational expression has the value 0.

The relational operators for Integer BASIC and Applesoft are the same, with one exception, as shown in Table 3-1.

TABLE 3-1. Relational Operators

Integer BASIC Operator	Operation	Applesoft Operation
<	Less Than*	<
>	Greater Than*	>
=	Equal To	=
#	Not Equal To	< > or > <
>=	Greater Than or Equal To*	> = or = <
<=	Less Than or Equal To*	< = or <
*Not allowed with strings in Integer BASIC.		

All relational operators have the same precedence; they are evaluated in order from left to right.

Here are some examples of relational expressions:

$1 = 5 - 4$	results in	1 (true)
$14 > 66$	results in	0 (false)
$15 > = 15$	results in	1 (true)
"AA" > "AA"	results in	0 (false)
"ANDERSON" < "ASHLEY"	results in	1 (true)
$(A = B) = (A\$ > B\$)$	depends	on the values of the variables. If the value of A is equal to the value of B and the value of A\$ is greater than the value of B\$, then this expression results in 1 (true).

The concept behind relational operators is easy enough to understand. The values 0 and 1 which BASIC arbitrarily assigns to false and true conditions can be used in integer and real expressions. This is not so easy to understand, since it is utterly arbitrary. For example, what meaning does the expression  $(1 = 1) * 4$  have? Outside of a BASIC program such an expression would be meaningless; but within BASIC  $(1 = 1)$  is true and true equates to 1, therefore the expression is the same as  $1 * 4$ , which results in 4. You can include relational expressions within other BASIC expressions. Here are some examples:

$25 + (14 > 66)$	is the same as	$25 + 0$
$(A + (1 = 5 - 4)) * (15 > = 15)$	is the same as	$(A + 1) * (1)$

## String Comparisons

You may be wondering what rules the Apple II uses when it compares strings. There are two considerations. First is string length. Strings of unequal length (remember blanks count toward string length) are not equal. If a shorter string is identical to the first part of a longer string, the longer string is greater than the shorter string (this is only pertinent in Applesoft). The second consideration is whether the strings contain the same characters, in the same order. In Integer BASIC, you can only compare two strings with the = and < operators. Strings are compared one character at a time, starting with the leftmost character — the first character of one string with the first character of the other, the second character with the second character, third with the third, and so on until one of the strings is exhausted or a character mismatch occurs.

Applesoft will compare the relative ordering of characters one by one. For comparison purposes, the letters of the alphabet have the order  $A > B$ ,  $B > C$ ,  $C > D$ , etc. Numbers that appear in strings have conventional ordering, namely  $0 > 1$ ,  $1 > 2$ ,  $2 > 3$ , etc. Other characters that appear in strings, like +, -, \$, and so on, are arbitrarily ranked in the order shown in Appendix I.

TABLE 3-2. Boolean Truth Table

The AND operation results in a 1 only if both values are 1.	
1 AND 1 = 1	1 AND 0 = 0
0 AND 1 = 0	0 AND 0 = 0
The OR operation results in a 1 if either value is 1.	
1 OR 1 = 1	1 OR 0 = 1
0 OR 1 = 1	0 OR 0 = 0
The NOT operation logically complements each value.	
NOT 1 = 0	
NOT 0 = 1	

### Boolean Expressions

Boolean operators give programs the ability to make logical decisions. Hence they are often called *logical operators*. There are four standard Boolean operators: AND, OR, Exclusive OR, and NOT. BASIC on the Apple II supports three of these operators: AND, OR, and NOT.

If you do not understand Boolean operators then a simple supermarket shopping analogy will serve to illustrate Boolean logic. Suppose you are shopping for breakfast cereals with two children. The AND Boolean operator says you will buy a cereal if both children select that cereal. The OR Boolean operator says that you will buy a cereal if either child selects it. The NOT Boolean operator generates an opposite. If child B insists on disagreeing with child A, then child B's decision is always the NOT of child A's decision.

Computers do not work with analogies; they work with numbers. Therefore Boolean logic reduces the values it operates on to 1 or 0 (true or false). Since Boolean operators work on the values 0 and 1, they are most often used with relational expressions (remember that relational expressions result in the value 0 or 1). Boolean operators can work on other types of operands, as we shall see shortly in the next section.

Table 3-2 summarizes the way in which Boolean expressions are evaluated. This table is referred to as a *truth table*. Boolean operators have equal precedence. If more than one Boolean operator is present in the same expression, they are evaluated from left to right. Here are some examples of Boolean expressions:

NOT ((3 + 4) >= 6)	results in	0 (false)
("AA" = "AB") OR ((8*2) = 4 ^ 2)	results in	1 (true)
NOT ("APPLE" = "ORANGE")		
AND (A\$ = B\$)	results in	1 (true) if A\$ and B\$ are equal; 0 (false) if not

## Mixed-Type Expressions

Very often expressions will not involve values of just one type. This is especially true in Applesoft, which has both integer and real numeric types. We have already introduced the idea of mixed-type expressions in our discussion of relational and Boolean expressions. You can mix types freely in any expression, except that strings cannot be part of integer, real, or Boolean expressions. Strings can only be present in string and relational expressions. Here are some examples of mixed-type expressions:

Legal	Illegal
3.1416 * (R ^ 2)	1600 + "PENNSYLVANIA AVENUE"
A% >= B/3	ST\$ < A%
43 AND 137	A\$ AND B\$
1 OR 4E + 10	NOT (A\$) = B\$
(A\$ = B\$) AND -6.25	NOT(A = B) OR C\$

The Apple II has several things to resolve when it evaluates a mixed-type expression. The first issue is the precedence of operators. Table 3-3 summarizes the operators for all types of expressions in order of precedence, from highest to lowest. This table shows us that anything in parentheses is evaluated first. If there is more than one level of parentheses present, the Apple II evaluates the innermost set first, then the next innermost, etc. (You will recall we covered this concept of *nesting* earlier.) Next, arithmetic expressions are evaluated. After that, relational expressions are evaluated. Finally, Boolean expressions are evaluated.

TABLE 3-3. Operators

	Precedence	Integer BASIC Operator	Applesoft BASIC Operator	Meaning
	High 9	()	()	Parentheses denote order of evaluation
Arithmetic Operators	8	^	^	Exponentiation
	7	-	-	Unary Minus
	6	*	*	Multiplication
	6	/	/	Division
	6	MOD	not available	Division Remainder**
	5	+	+	Addition
	5	-	-	Subtraction
Relational Operators	4	=	=	Equal
	4	#	<> or ><	Not equal
	4	<	<	Less than
	4	>	>	Greater than
	4	<=	<= or = <	Less than or Equal
	4	>=	>= or = >	Greater than or Equal
Boolean Operators	3	NOT	NOT	Logical complement
	2	AND	AND	Logical AND
	1	OR	OR	Logical OR
	Low			
** Integer BASIC only				

As we noted earlier, relational expressions return a value of 0 or 1 depending on whether the relationship being tested is false or true. Thus, a relational expression can exist as part of an integer or real expression.

You can also mix types in Boolean expressions. Everything in a Boolean expression is converted to 0 or 1 before any Boolean operations take place. Numeric values are converted according to the following rule: if the value is 0, it remains zero; any non-zero value is converted to 1.

BASIC cannot automatically convert strings to numeric values, so strings are illegal in integer, real, and Boolean expressions, except as part of a relational expression.

In Applesoft both integer and real values can be present in the same real, relational, or Boolean expression.

Whenever they occur in a real expression, integer values are converted to real values temporarily in order to evaluate the expression. The final result of such an expression can be either an integer value or a real value, depending on the context in which the expression occurs. Applesoft will convert the value automatically as appropriate.

Real values are converted to integers by discarding the fractional part and using the next lower whole number. This is called *truncation*. For example:

1.1	becomes	1
1.9	becomes	1
-1.1	becomes	-2
-1.9	becomes	-2

## BASIC STATEMENTS

We are now ready to describe the part of the BASIC statements which specifies the operation the statement will perform. You specify these operations with BASIC instructions. It is common practice to use the terms *statement* and *command* interchangeably and somewhat ambiguously. Strictly speaking, a command is an instruction issued in immediate mode. The same instruction in programmed mode is a statement.

Each instruction performs a specific task. This chapter introduces you to programming concepts, stressing the way statements are used. We do not describe statements in detail in this chapter. Be sure to read the complete statement description given in Chapter 8 to get a thorough understanding of how a statement works. This chapter will give you an understanding of only part of a statement's total capabilities.

One last caveat before we begin. Although this chapter introduces you to programming concepts, it cannot possibly cover programming in depth. If you want or need more instruction in programming, consult one of the BASIC primers listed in Appendix K.

## REMARKS

It is appropriate that any discussion of BASIC statements begins by describing the only BASIC statement which the computer will ignore: the remark. If the first three characters of a BASIC statement are REM, then the computer ignores the statement entirely. So why include such a statement? The answer is that remarks make your program easier to read.

If you write a short program with five or ten statements, you will probably have little trouble remembering what the program does — unless you leave it around for six months and then try to use it again. If you write a longer program with 100 or 200 statements, then you are quite likely to forget something very important about the program the very next time you use it. After you have written dozens of programs, you will stand no chance of remembering each program in detail. The solution to this problem is to document your program by including remarks that describe what is going on.

Good programmers use plenty of remarks in all of their programs. In all of this chapter's program examples we will include remarks that describe what is going on, simply to get you into the habit of doing the same thing yourself.

Remark statements have line numbers, like any other statement. A remark statement's line number can be used like any other statement's line number.

## ASSIGNMENT STATEMENTS

Assignment statements let you assign values to variables. You will encounter assignment statements frequently, in every type of BASIC program.

Here is an example of an assignment statement:

```
90 REM INITIALIZE VARIABLE X
100 LET X=3
```

In statement 100, variable X is assigned the value 3; this same statement could be rewritten:

```
100 X=3
```

The word LET is optional; usually it is omitted.

Here is a string variable assignment statement:

```
215 A$="ALSO RAN"
```

The string variable A\$ is assigned the two words ALSO RAN.

Here are three assignment statements that assign values to array variable R\$(), which we encountered earlier when describing arrays:

```
200 REM R$() IS THE MOTEL GUEST LIST
210 R$(1)="JONES"
220 R$(2)="SMITH"
230 R$(3)="DOE"
```

Remember, we can put more than one statement on a single line; therefore the

three R\$ assignments could be placed on a single line as follows:

```
200 REM   R$( ) IS THE MOTEL GUEST LIST
210 R$(1)="JONES":R$(2)="SMITH":R$(3)="DOE"
```

Recall that a colon must separate adjacent statements appearing on the same line.

Assignment statements can include any of the arithmetic or logical operators described earlier in this chapter. Here is an example of such an assignment statement:

```
90 REM   THIS A DUMB WAY OF ASSIGNING A VALUE
100 V=33+7/9
```

The statement above assigns the value 4.17647059 to real variable V; it is equivalent to these three statements:

```
90 REM   X AND Y NEED TO BE INITIALIZED SEPARATELY FOR
        LATER USE
100 X=7
110 Y=9
120 V=33+X/Y
```

which could be written on one line as follows:

```
100 X=7:Y=9:V=33+X/Y
```

Here are assignment statements that perform the Boolean operations given earlier in this chapter:

```
90 REM   THESE EXAMPLES WERE DESCRIBED EARLIER IN THE
        CHAPTER
100 A= NOT ((3+4)>=6)
110 B=("AA"="AB") OR ((8*2)=(4 ^ 2))
```

The following example shows how a string variable could have its value assigned using string concatenation in Applesoft:

```
90 REM   R$(6) IS ASSIGNED THE VALUE MR. ALTON
100 MR$ = "MR. "
110 MS$ = "MS. "
120 N$ = "ALTON"
200 R$(6) = MR$ + N$
```

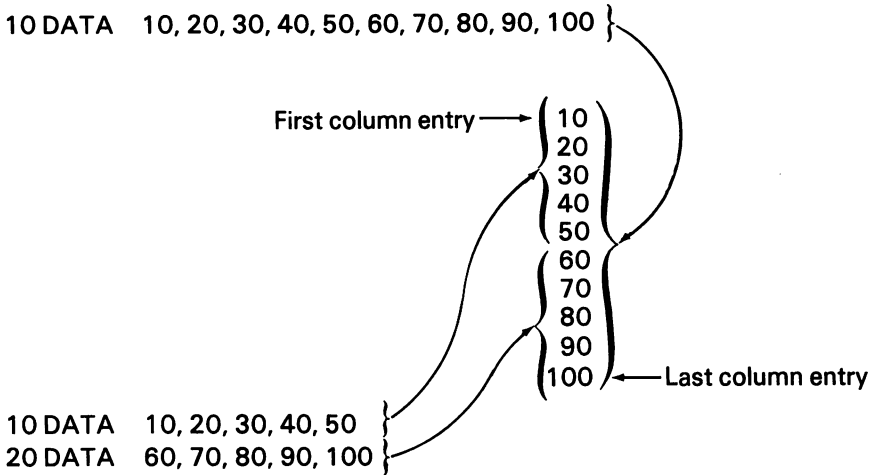
## DATA and READ Statements

When a number of variables need data assignments in an Applesoft program, you can use the DATA and READ statements rather than the previous type of assignment statement. Consider the following example:

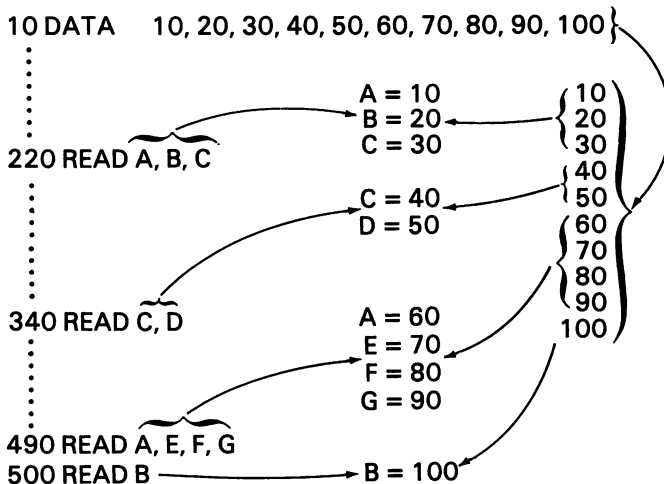
```
5 REM   INITIALIZE ALL PROGRAM VARIABLES
10 DATA 10, 20, -4, 300
20 READ A,B,C,D
```

The statement on line 10 specifies four numeric data values. These four values are assigned to four numeric variables by the statement on line 20. After the statements on lines 10 and 20 have been executed, A=10, B=20, C=-4, and D=300.

If you have one or more DATA statements in your program, then you can visualize them as building a column of values. For example, a DATA statement that contains a list of ten values would build a ten-entry column. Two DATA statements each specifying five of the ten data entries would build exactly the same column. This may be illustrated as follows:



The first READ statement in the program starts at the first column entry and takes values sequentially, assigning them to variables named in the READ statement. The second (and subsequent) READ statements take values from the column, starting at the point where the previous READ statement left off. This may be illustrated as follows:

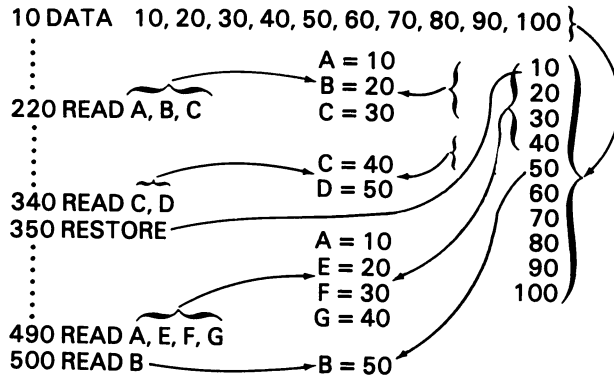


The DATA column can contain both numeric and string values. When you assign the values to variables using a READ statement, each variable must be the same type (string or numeric) as the corresponding value it is assigned.



### RESTORE Statement

You can at any time send the pointer back to the beginning of the DATA column by executing a RESTORE statement in Applesoft. Here is how RESTORE works:



### Clearing Variables

Both Integer BASIC and Applesoft let you set every numeric variable and array element to zero, and every string variable and array element to null, all at once.

The CLR command does this in Integer BASIC. You can use it only in immediate mode. Here is an example:

```

>X=37

>PRINT X
37

>CLR

>PRINT X
0
  
```

The CLEAR statement does this in Applesoft. It also resets the DATA column pointer like RESTORE does. Here is an example:

```

110 REM INITIALIZE VARIABLES
120 X=37
130 A$="PIG IRON"
140 PRINT A$
150 CLEAR
160 PRINT X

1RUN
PIG IRON
0
  
```

## DECLARING ARRAY AND STRING SIZE

If you plan to use arrays or string variables in your program, you need to declare their maximum sizes (or dimensions) in a DIM statement or statements (called dimension statements) at the beginning of the program. A dimension statement can provide dimensions for any number of arrays and string variables as long as the statement fits on a standard program line.

In Integer BASIC, you do this by stating the array or string variable name and then specifying its maximum size, enclosed in parentheses. Only one-dimensional numeric arrays are allowed — no string arrays or multiple-dimensioned arrays. The following example dimensions two strings of 5 and 25 characters respectively, and a numeric array of 13 elements (0 through 12):

```
10 DIM S1$(5),S2$(25),NB(12)
```

The number following a string variable name in a DIM statement is the maximum length that string can be during the program. The number following a numeric array name is equal to the largest index value that you can use for that array.

The first time Applesoft encounters an array, it checks to see if you have dimensioned it. If not, it automatically dimensions the array with indexes 0 through 10 for each dimension used. So you need not dimension an array unless you want more than 11 indexes in any one dimension. The following example dimensions the single indexed array R\$. It also dimensions an integer array of 21 elements.

```
115 DIM R$(10),RZ(20)
```

The double-dimension hotel guest list variable from our earlier example of arrays would be dimensioned as follows:

```
115 DIM H$(8,10)
```

The number (or numbers) following an array name in a DIM statement is equal to the largest index value that can occur in that particular index position. But remember that indexes begin at 0. Therefore R\$(10) dimensions the variable R\$( ) to have 11 values, not 10, since indexes 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 will be allowed. H\$(8,10), likewise, specifies a doubly-dimensioned variable with 99 entries, since the first dimension can have values 0, 1, 2, 3, etc., while the second dimension can have values 0 through 10.

## Redimensioning Arrays

Once you have dimensioned an array variable you cannot redimension it without rerunning the whole program. Subsequent references cannot use an index higher than the number of indexes you declared; each index must have a value between 0 and the number of indexes dimensioned.

## BRANCH STATEMENTS

Statements within a BASIC program are normally executed in ascending order of line numbers. This execution sequence was explained earlier in this chapter when we described line numbers. Branch statements change this execution sequence.

### GOTO Statement

GOTO is the simplest branch statement; it allows you to specify the statement which will be executed next. Consider the following example:

```
20  A = 4
30  GOTO 100
40
50
60
70
80
90
100
110
etc.
```

The statement on line 20 is an assignment statement; it assigns a value to variable A. The next statement is a GOTO; it specifies that program execution must branch to line 100. Therefore the instruction execution sequence surrounding this part of the program will be: line 20, then line 30, then line 100.

Of course, some other statement must branch back to line 40, otherwise the statement on line 40 would never be executed by program logic as illustrated above.

You can branch to any line number, even if the line has nothing but a remark on it. However, the computer ignores the remark, so the effect is the same as branching to the next line. For example, consider the following branch:

```
20  A = 4
30  GOTO 70
40
50
60
70  REM THIS LINE CONTAINS ONLY A REMARK
80
90
etc.
```

Program execution branches from line 30 to line 70; there is nothing but a remark on line 70, therefore the computer moves on to line 80, executing statements on this line. Therefore, even though you can branch to a remark, you might as well branch to the next line. This may be illustrated as follows.

```

20  A = 4
30  GOTO 80
40
50
60
70  REM THIS LINE CONTAINS ONLY A REMARK
80
90
etc.

```

A diagram showing a curved arrow starting from the right side of line 30 and pointing to the right side of line 80, illustrating the jump performed by the GOTO statement.

Attempting to branch to a nonexistent line number causes an error message.

### Computed GOTO Statement

There is also a kind of GOTO statement that lets program logic branch to one of two or more different line numbers, depending on the current value of a numeric expression.

Consider the Integer BASIC statement sequence:

```

10
20
30  A = B - 3
40  GOTO 30*A + 50
50
60
70
80
90
100
110
120
etc.

```

A diagram showing a curved arrow starting from the right side of line 40 and branching to the right sides of lines 50, 80, and 110. To the left of these lines are labels: 'A = 0' next to line 50, 'A = 1' next to line 80, and 'A = 2' next to line 110, indicating the conditional branching based on the value of variable A.

The statement on line 40 is a *computed* GOTO. When this statement is executed, program logic will branch to the statement number computed by evaluating the expression. In this example it branches to statement 50 if variable  $A=0$ , to statement 80 if  $A=1$ , while  $A=2$  causes a branch to statement 110. If the computed line number does not exist in the program a \*\*\* BAD BRANCH ERR message results. Notice that variable A is assigned a value in statement 30. The value assigned to A depends on the current value of variable B. The illustration does not show how variable B is computed; however, so long as B has a value of 3, 4, or 5, the statement on line 40 will cause a branch to occur.

To test the computed GOTO statement in Integer BASIC, key in the following program:

```

>9  REM INITIALIZE VARIABLE B
>10 B = 4
>20 PRINT B
>30 A = B - 3

```

```

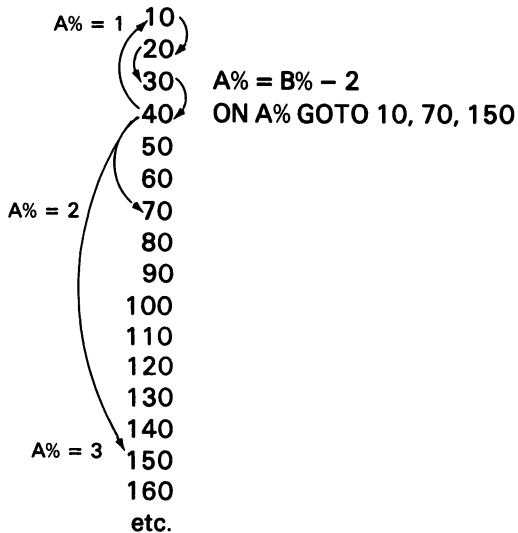
>40  GOTO 30 * A + 50
>49  REM B = 3
>50  END
>79  REM B = 4
>80  PRINT B
>90  B = 5
>100 GOTO 20
>109 REM B = 5
>110 PRINT B
>120 B = 3
>130 GOTO 20

```

Now execute this program by typing RUN.

Can you account for the sequence in which digits are displayed? Try re-writing the program so that each number is displayed once, in the sequence: 345345345. . .

Applesoft has a slightly different version of the computed GOTO statement, as shown below.



The statement on line 40 is the Applesoft form of computed GOTO. When this statement is executed, program logic will branch to statement 10 if variable  $A\% = 0$ , to statement 70 if variable  $A\% = 2$ , while  $A\% = 3$  causes a branch to statement 110. If  $A\%$  has any value other than 1, 2 or 3, the program continues at statement 50.

The expression in an Applesoft computed GOTO statement is evaluated and its value determines which line number to branch to from the computed GOTO list of line numbers. If the value is 1 the first line number is used, if the value is 2 the second line number is used, and so on. If the value is 0, or exceeds the number of line numbers in the list, the program falls through to the statement immediately following the computed GOTO statement.

The following Applesoft program demonstrates how the computed GOTO statement works.

```
10 B% = 4
20 PRINT B%
30 A% = B% - 2
40 ON A% GOTO 10,70,150
70 PRINT B%
80 B% = 5
90 GOTO 30
150 PRINT B%
160 B% = 3
170 GOTO 20
```

## LOOPS

GOTO and computed GOTO statements let you create any sequence of statement execution that your program logic may require. But suppose you want to reexecute an instruction (or a group of instructions) many times. For example, suppose array variable A(I) has 100 elements and each element needs to be assigned a value ranging from 0 to 99. Writing 100 assignment statements would be very tedious. It is far simpler to reexecute one statement 100 times in a loop.

### FOR and NEXT Statements

You can create a loop using the FOR and NEXT statements as follows:

```
10 DIM A(99)
20 FOR I=0 TO 99 STEP 1
30 A(I)=I
40 NEXT I
```

Statement(s) between FOR and NEXT are executed repeatedly. In this case a single assignment statement appears between FOR and NEXT; therefore this single statement is reexecuted repeatedly. This kind of program structure is called a *FOR-NEXT loop*.

So you can see the workings of FOR-NEXT loops, the following program displays the values it assigns to array A( ) within the loop.

```
10 DIM A(99)
20 FOR I=0 TO 99 STEP 1
30 A(I)=I
35 PRINT A(I)
40 NEXT I
50 END
```

When you key in RUN, the program displays 100 numbers, starting at 0 and ending at 99.

Statements between FOR and NEXT are reexecuted the number of times specified by the *index variable* appearing directly after FOR; in the illustration above this index variable is I. I is specified as going from 0 to 99 in *steps* of 1.

Variable *I* also appears in the assignment statement on line 30. Therefore the first time the assignment statement is executed, *I* will equal 0 and the assignment statement will be executed as follows:

```
30 A(0)=0
```

*I* is increased by the step size, which is specified on line 20 as 1; *I* therefore equals 1 the second time the assignment statement on line 30 is executed. The assignment statement has effectively become:

```
30 A(1)=1
```

*I* continues to be incremented by the specified step until the maximum value of 99 is reached (or exceeded).

The step does not have to be 1; it can have any integer value. Change the step to 5 on line 20 and reexecute the program. Now the assignment statement is executed just 20 times, since incrementing *I* 19 times by 5 will take it to 95; the 20th increment will take it to 100, which is more than the maximum value of 99. Keeping the step at 5, we could allow the assignment statement to be executed 100 times by increasing the maximum value of *I* to 500. Can you make this change? (Remember to change the DIM statement as well.)

The step size does not have to be positive. But if the step size is negative, then the initial value of *I* must be larger than the final value of *I*. For example, if the step size is -1 and we want to initialize 100 elements of *A(I)* with values ranging from 0 to 99, then we would have to rewrite the statement on line 20 as follows:

```
10 DIM A(99)
20 FOR I=99 TO 0 STEP -1
30 A(I)=I
35 PRINT A(I)
40 NEXT I
50 END
```

Execute this program to test the negative step.

If the step size is 1 (and this is frequently the case), you do not have to specify a step size definition. In the absence of any definition, BASIC assumes a step size of 1.

You may specify the initial and final index values and the step size using expressions if you wish. But you should avoid doing so, since this unnecessarily complicates the program. If you must calculate one of these values, it is more efficient to do so in a separate statement ahead of the loop.

You can use real values for the initial and final index values and for the step size in Applesoft. You do not need to specify the index variable in the NEXT statement in Applesoft. But if you do, it will make your program easier to read.

### Nested Loops

The FOR-NEXT structure is referred to as a *program loop* since statement execution loops around from FOR to NEXT, and back to FOR. This loop structure is very


common; almost every BASIC program that you write will include one or more such loops. Loops are so common that they are frequently *nested* one inside the other like a set of mixing bowls. There can be any number of statements between FOR and NEXT. Frequently there are tens, or even hundreds of statements. And within these tens or hundreds of statements additional loops may occur. The following illustration shows a single level of nesting:

```

10 DIM A(99)
20 FOR I=0 TO 99 STEP 1
30 A(I)=I
40 REM DISPLAY ALL VALUES OF A(I) ASSIGNED THUS FAR
50 FOR J=0 TO I
60 PRINT A(J)
70 NEXT J
80 NEXT I
90 END

```

Complex loop structures appear frequently, even in relatively short programs. Here is an example, showing the FOR and NEXT statements, but none of the intermediate statements:



```

50 FOR I=1 TO 10
60 FOR X=25 TO 347 STEP 3
:
:
100 FOR A=9 TO 0 STEP -1
:
:
140 NEXT A
200 FOR B=25 TO 100 STEP 5
:
:
280 NEXT B
300 NEXT X
:
:
500 FOR Y=1 TO 20 STEP 2
:
:
600 FOR P=10 TO 20
:
:
650 NEXT P
700 NEXT Y
:
:
1000 FOR Z=1 TO 10
:
:
1090 NEXT Z
:
:
1200 NEXT I
:
:

```



The outermost loop uses index variable I; it contains three nested loops that use indexes X, Y, and Z. The X-loop contains two additional loops which use indexes A and B. The Y-loop contains one nested loop using index P. The Z-loop contains no nested loops.

Loop structures are very easy to visualize and use. There is only one common error which you must avoid: do not terminate an outer loop before you terminate an inner loop. For example, the following loop structure is illegal:

```

50 FOR I=1 TO 10
60 FOR X=25 TO 347 STEP 3
   :
   :
100 NEXT I
   :
   :
200 NEXT X

```

Every program must have the same number of FOR and NEXT statements, since every loop must begin with a FOR statement and end with a NEXT statement.

For example, suppose there is one FOR statement, but two NEXT statements. The first NEXT statement terminates a FOR statement so the loop will execute correctly. But the second NEXT statement has no FOR statement, which causes an error.

If you do not include the index variable in the NEXT statement in an Applesoft program, then program logic will automatically terminate loops correctly, since there is only one possible correct loop termination each time a NEXT statement is encountered. If you do not believe this, look again at the complex example illustrated previously. Then work out some additional complex examples.

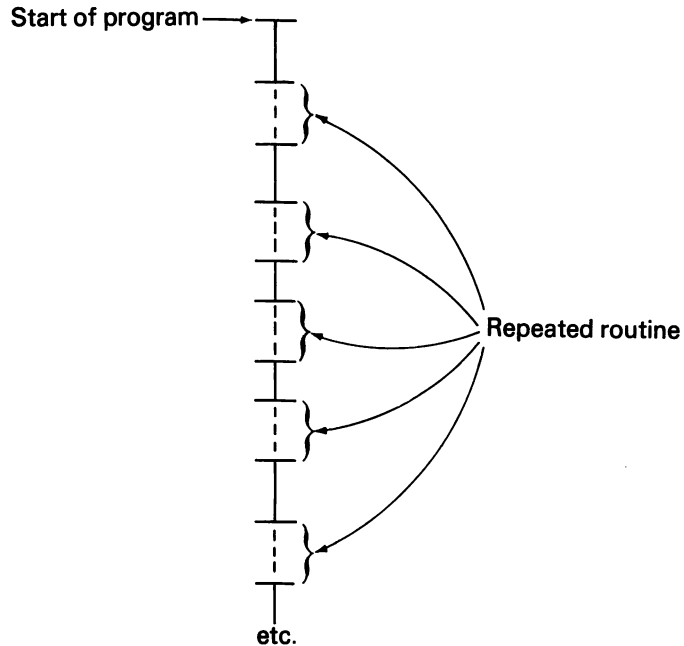
## SUBROUTINE STATEMENTS

Once you start writing programs that are more than a few statements long, you will quickly find short sections of program that get used repeatedly. For example, suppose you have an array variable (such as A()) which is reinitialized frequently at different points in your program. Would you simply repeat the three instructions that constitute the FOR-NEXT loop that we described earlier? Since there are just three instructions, you may as well do so.

But suppose the loop has ten or eleven instructions that process array data in some fashion before it initializes the array. If you had to use this loop many times within one program, rewriting the same ten to fifteen statements each time you wished to use the loop would take time, but more importantly it would waste a lot of computer memory. This concept is illustrated at the top of the following page.

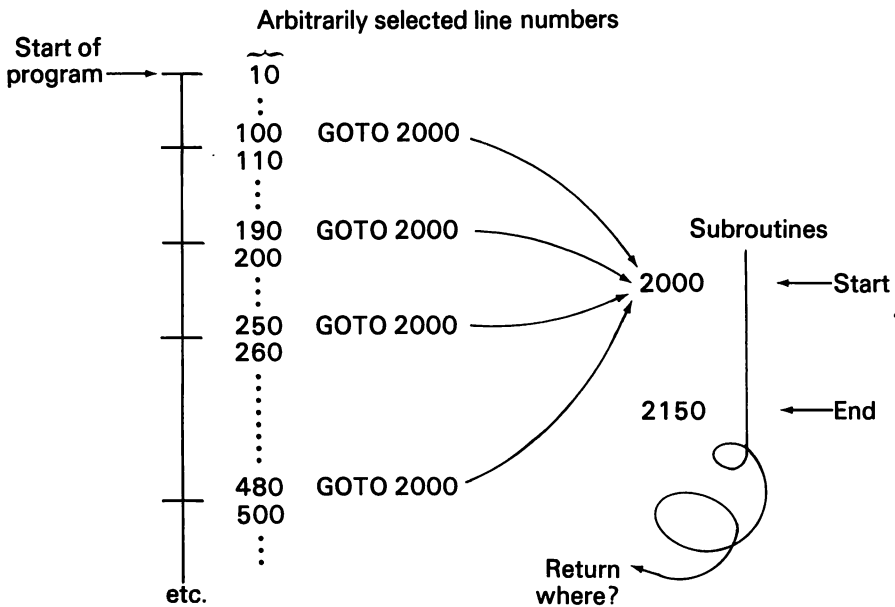
How about separating out the repeated statements and branching to them?

That is precisely what we will do; the group of statements is then referred to as a *subroutine*.



But a problem arises. Branching from your program to the subroutine is simple enough; the subroutine has a specific starting line number. But at the end of the subroutine, where do you branch back to?

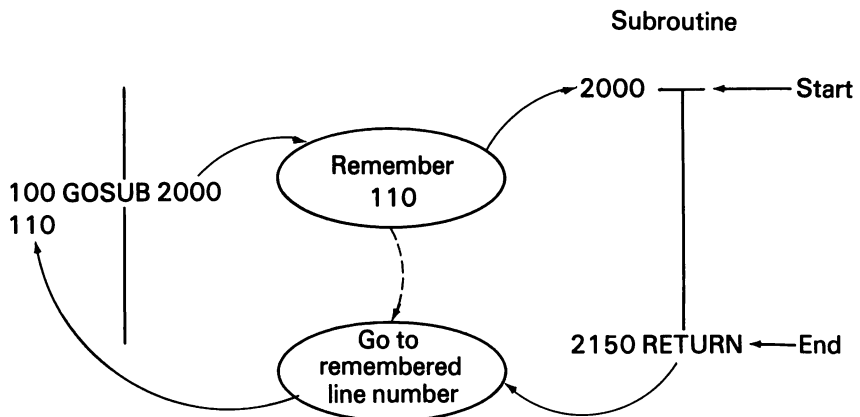
You can execute a GOTO statement whenever you wish to branch to a subroutine. This may be illustrated as follows:



But at the end of the subroutine, where do you return to? If two or more GOTO statements branch to the subroutine, there are two or more different places to which you will wish to return after the subroutine has completed execution. The solution is to use special subroutine statements. Instead of branching to the subroutine using a GOTO, use a GOSUB statement.

### GOSUB Statement

This statement branches in the same way as a GOTO, but in addition it remembers where to return to. In computer jargon, we say GOSUB *calls* a subroutine. This may be illustrated as follows:



End the subroutine with a RETURN statement. This statement causes a branch back to the statement following the GOSUB statement. If the GOSUB is the last statement on the line, the program returns to the first statement on the next line.

The three-statement loop which initializes array A(), if it were converted into a subroutine, would appear as follows:

```

10 REM  MAIN PROGRAM
20 REM  YOU CAN DIMENSION A SUBROUTINE'S
30 REM  VARIABLE IN THE MAIN PROGRAM.
40 REM  IT IS A GOOD IDEA TO DIMENSION ALL
50 REM  VARIABLES AT THE START OF THE MAIN PROGRAM
60 DIM A(99)
70 GOSUB 2000
80 REM  DISPLAY SOMETHING TO PROVE THE RETURN OCCURRED
90 PRINT "RETURNED"
100 END
200 NEXT I
2000 REM  SUBROUTINE
2010 FOR I=0 TO 99
2020 A(I)=I
2030 PRINT A(I)
2040 NEXT I
2050 RETURN

```

## POP Statement

Under some circumstances you will not want a subroutine to return to the statement following the GOSUB statement. You might be tempted to just use a GOTO statement to return, but that can cause a problem because BASIC is still remembering where it should return to. In cases like this, use the POP statement. Otherwise you risk an error caused by the accumulation of unused RETURN statements. All POP does is make BASIC forget the most recent return location. You can then use a GOTO statement to branch somewhere else in the program.

Bypass the RETURN statement sparingly. Using POP excessively to enable GOTO branching out of subroutines leads to tangled, confusing programs.

## Nested Subroutines

Subroutines can be nested. That is to say, a subroutine can itself call another subroutine, which in turn can call a third subroutine, and so on. You do not have to do anything special in order to use nested subroutines. Simply branch to the subroutine using a GOSUB statement and end the subroutine with a RETURN statement. BASIC will remember the correct line number for each nested return.

The following program illustrates nested subroutines:

```
10 REM   MAIN PROGRAM
20 REM   YOU CAN DIMENSION A SUBROUTINE'S
30 REM   VARIABLE IN THE MAIN PROGRAM.
40 REM   IT IS A GOOD IDEA TO DIMENSION ALL
50 REM   VARIABLES AT THE START OF THE MAIN PROGRAM
60 DIM A(99)
70 GOSUB 2000
80 REM   DISPLAY SOMETHING TO PROVE THE RETURN OCCURRED
90 PRINT "RETURNED"
100 END
2000 REM  FIRST LEVEL SUBROUTINE
2010 FOR I=0 TO 99
2020 A(I)=I
2030 GOSUB 3000
2040 NEXT I
2050 RETURN
3000 REM  NESTED SUBROUTINE
3010 PRINT A(I)
3020 RETURN
```

This program moves the PRINT A(I) statement out of the subroutine at line 2000 and puts it into a nested subroutine at line 3000. Nothing else changes.

While it is perfectly acceptable and even desirable for one subroutine to call another, a subroutine cannot call itself. Neither can a subroutine call another subroutine which in turn calls the first subroutine. This is called *recursion*, and is not allowed in BASIC on the Apple II.

### Computed GOSUB Statement

GOTO and GOSUB statement logic is very similar. The only difference is that GOSUB remembers the next line number. It will therefore not come as any surprise that there is a computed GOSUB statement akin to the computed GOTO statement. The computed GOSUB statement allows you to branch to one of two or more subroutines depending on the value of a numeric expression. The computed GOSUB statement remembers where to return to. It does not matter which of the subroutines gets called, the called subroutine's RETURN statement will cause a branch back to the remembered line number.

You can nest subroutines using computed GOSUB statements, just as you can nest subroutines using standard GOSUB statements.

Consider the following Integer BASIC statement:

```
>100 GOSUB A*500+2000
>110 REM
```

The expression on line 100 is a computed GOSUB. When this statement is executed, program logic branches to the subroutine at the line number computed by evaluating the expression. In this example, it branches to statement 2000 if A=0, to statement 2500 if A=1, and so on. If the computed line number does not exist in the program a \*\*\* BAD BRANCH ERR message results.

The Applesoft version of the computed GOSUB statement works in a manner similar to the Applesoft computed GOTO statement. Here is an example:

```
190
1100 ON A GOSUB 1000,500,5000,2300
1110 REM
```

When the statement on line 100 is executed, if A=1 the subroutine beginning at line 1000 is called. If A=2 the subroutine beginning at line 500 is called. If A=3 the subroutine beginning at line 5000 is called. If A=4 the subroutine beginning at line 2300 is called. If A has any value other than 1, 2, 3, or 4, program execution falls through to line 110 (no subroutine is called).

### CONDITIONAL EXECUTION

The computed GOTO and computed GOSUB are conditional statements. That is, the exact flow of program execution depends on the values of one or more variables which can change as the program is running. The exact program flow depends on the condition of the variables.

#### IF-THEN Statements

Another conditional statement is the IF-THEN statement. It has the general form:

*IF expression THEN statement*

If the *expression* is true, then the *statement* is executed. Relational and Boolean expressions are most common with IF-THEN statements, but arithmetic expressions can be used as well. This gives a BASIC program real decision-making capabilities. Here are three simple examples of IF-THEN statements:

```
10 IF A=B+5 THEN PRINT MSG$
40 IF CC$="M" THEN IN=0
50 IF Q<14 AND M<M1 THEN GOTO 66
```

The statement on line 10 causes a PRINT statement to be executed if the value of variable A is five more than the value of variable B. The PRINT statement will not be executed otherwise.

The statement on line 40 sets numeric variable IN to 0 if string variable CC\$ is the letter M.

The statement on line 50 causes program execution to branch to line 66 if variable Q is less than 14, and variable M is less than variable M1. Otherwise program execution will continue with the statement on the next line.

If you do not understand the evaluation of expressions following IF, then refer to the discussion of expressions given earlier in this chapter.

An IF-THEN statement can be followed by other statements on the same program line. Integer BASIC and Applesoft handle this situation somewhat differently.

In Integer BASIC, only the statement which immediately follows THEN is conditionally executed. Any later statements on the same program line are always executed, no matter whether the expression in the IF-THEN statement is true or false. This may be illustrated as follows:

```
10 IF V>100 THEN PRINT "DEWEY WINS": GOSUB 2000
20 T=T+V: PRINT T
```

In the example above, the program will only print the message DEWEY WINS if the value of variable V is greater than 100. The program will call the subroutine at line 2000 no matter what the value of V is.

Applesoft executes statements that follow an IF-THEN statement on the same line only if the expression in the IF-THEN statement is true. If the expression is false, program execution drops down to the first statement on the next program line. In the example above, the program will print the message DEWEY WINS and call the subroutine at line 2000 only if the value of variable V is greater than 100. If V is less than or equal to 100, the program will not print the message *or* call the subroutine, but will instead proceed directly to the first statement on line 20.

There is a special form of the IF-THEN statement available in Applesoft. Whenever the conditionally executed statement is a GOTO statement, you can omit the word THEN if you wish. The following two statements are equivalent:

```
110 IF MM$=DD$ THEN GOTO 100
```

is the same as:

```
110 IF MM$=DD$ GOTO 100
```

## INPUT AND OUTPUT STATEMENTS

There are a variety of BASIC statements that control the transfer of data to and from the computer. Collectively these are referred to as *input/output statements*. The simplest input/output statements control data input from the keyboard and data output to the display screen. We are going to discuss these simple input/output statements in the paragraphs that follow. But there are also more complex input/output statements that control data transfer between the computer and peripheral devices such as cassette recorder, disk drives, and printers. These more complex input/output statements are described in Chapters 4 and 5. Chapter 6 covers output statements to the display screen for graphics.

We have already encountered the PRINT statement, which outputs data to the display screen. So let's discuss this statement first, before looking at input statements.

### PRINT Statement

Why use PRINT instead of DISPLAY or some abbreviation of the word display? The answer is that in the early sixties, when the BASIC programming language was being created, displays were very expensive and generally unavailable on medium or low cost computers. The standard computer terminal had a keyboard and a printer. Information was printed where today it is displayed; hence the use of the word PRINT to describe a statement which causes a display.

The PRINT statement will display text or numbers. For example, the following statement will display the single word TEXT:

```
10 PRINT "TEXT"
```

To display a number, you place the number, or a variable name, after PRINT, like this:

```
>A=10
```

```
>PRINT 5,A
5      10
```

The statement above displays the number 5, and then the number 10 on the same line.

You can display a mixture of text and/or numbers by listing the information to be displayed after PRINT. Use commas to separate individual items. The following PRINT statement displays the words ONE, TWO, THREE, FOUR, and FIVE, followed by the numeral for each number:

```
10 PRINT "ONE",1,"TWO",2,"THREE",3,"FOUR",4,"FIVE",5
20 END
```

If you separate variables with commas, as we did above, then the Apple II automatically allocates a fixed number of spaces for each variable displayed. Try executing the statement illustrated above in immediate mode to prove this to yourself. If you want the display to take out empty spaces, separate the variables

using semicolons, as follows:

```
10 PRINT "ONE";1;"TWO";2;"THREE";3;"FOUR";4;"FIVE";5
20 END
```

Again enter this statement in immediate mode to understand how the semicolon works.

A PRINT statement will automatically return the cursor to the left margin and drop it down one line as its last action. In computer jargon, this is called a *carriage return*. You can suppress the carriage return by putting a comma or a semicolon after the last value in the PRINT list. A comma occurring after the last value will move the cursor to where the next value would be displayed, if there were one. To illustrate this, type in the following three-statement program and run it by typing in RUN:

```
10 PRINT "ONE",1,"TWO",2
20 PRINT "THREE",3,"FOUR",4
30 END
```

Now add a comma to the end of the statement on line 10 and again execute the program by typing RUN. You will see the two lines of display occur on a single line.

Now replace the comma at the end of line 10 with a semicolon and again run the program. The display occurs on a single line, but the space between the numeral 2 and the word THREE has been removed. By changing other commas to semicolons you can selectively remove additional spaces.

We have been displaying the numerals by inserting them directly into the PRINT statement. You can if you wish display the contents of variables instead. The following program does the same thing as the first PRINT statement example, but uses array A( ) to create digits. Try entering this program and running it:

```
5 DIM A(5)
10 FOR I=1 TO 5
20 A(I)=I
30 NEXT I
40 PRINT "ONE";A(1);"TWO";A(2);"THREE";A(3);"FOUR";A(4);
   "FIVE";A(5)
50 END
```

In Applesoft, you can put the displayed words into a string variable and move the PRINT statement into a FOR-NEXT loop by changing the program as follows:

```
10 DATA "ONE","TWO","THREE","FOUR","FIVE"
20 FOR I = 1 TO 5
40 READ N$
50 PRINT N$;I;
60 NEXT I
70 END
```

## INPUT Statement

When an INPUT statement is executed, the computer waits for input from the keyboard; until the computer gets the input it requires, nothing else will happen.



In its simplest form, an INPUT statement begins with the word INPUT and is followed by a variable name. Data entered from the keyboard is assigned to the named variable. The variable name type determines the type of data which must be entered. A numeric variable name can be satisfied only by numeric input. To demonstrate numeric input, key in the following short program and run it (try entering some alphabetic data and see what happens):

```
10 INPUT A
20 PRINT A
25 REM END PROGRAM IF 0 ENTERED
30 IF A = 0 THEN END
40 GOTO 10
```

Upon executing an INPUT statement, the computer displays a question mark, then waits for your entry. The program above displays each key as you press it. In computer jargon, the display screen *echoes* the keyboard. In addition, the number is displayed again because of the PRINT statement on line 20. The first display occurs when the INPUT statement on line 10 is executed and you make an entry at the keyboard. The second display is in response to the PRINT statement on line 20.

An INPUT statement can input more than one value at a time. To do this, list all the variables you want to input values for following the word INPUT. Separate the variables with commas. When such an INPUT statement is executed, you must respond with a separate value for each variable. Be sure each value is the same type as the variable it will be assigned to.

When you respond to an INPUT statement, do not use commas as punctuation in large numbers; enter 1000 *not* 1,000.

The following example inputs two numeric values then displays these inputs.

```
20 INPUT A,B
30 PRINT A,B
35 REM END PROGRAM IF 0 ENTERED
40 IF A=0 OR B=0 THEN END
50 GOTO 20
```

Run the program above and try entering one number followed by a comma, after that another number, and then press RETURN. Now try something a bit different. Enter one number and press RETURN. As you see, the Apple II reminds you to enter the next value. So enter another number and press RETURN. Thus, when an INPUT statement calls for more than one numeric value, you have a choice of entering all the values on one line or entering them on separate lines.

The INPUT statement works somewhat differently with string variables in Integer BASIC. First of all, it does not display a question mark. Try this example:

```
10 DIM A$(19)
20 INPUT A$
30 PRINT A$
35 REM END PROGRAM IF NULL ENTRY
40 IF A$="" THEN END
50 GOTO 20
```

When you run the program above, try entering a string of more than ten characters. You will get a \*\*\* STR OVFL ERR message and the program will stop. The length of the string you enter cannot exceed the maximum length of the string variable used in the INPUT statement.

Integer BASIC forces you to enter each string value on a separate line. If an INPUT statement specifies a list of variables, and there are string variables on the list, the associated string values must be entered on separate lines. This is because Integer BASIC lets you include commas as part of a string value. You can prove this for yourself by running the example program above and entering the string value DOE, JOHN. The following example illustrates what happens when a string variable is part of an INPUT statement in Integer BASIC. Experiment with this program; try to enter all four values on the same line, separated by commas. What happens? Try entering each value on a separate line. See what happens if you enter a numeric value or a comma as part of a string value.

```
10 DIM A$(10),B$(10)
20 INPUT A$,A,B$,B
30 PRINT A$,A,B$,B
35 REM END PROGRAM IF NULL ENTRY
40 IF A$="" THEN END
50 GOTO 20
```

As we discussed earlier, any real variable can have an integer value in Applesoft. Therefore you can input an integer value for a real variable. A real value entered for an integer variable is converted to an integer value according to the truncation rules presented in the "Mixed-Type Expressions" section of this chapter.

### INPUT Statement Prompts

The INPUT statement is very fussy; its syntax is too demanding for any normal human operator. Just imagine some poor office worker who knows nothing about programming; on encountering the types of error message which can occur if one comma happens to be out of place, he will give up in despair. You are therefore likely to spend a lot of time writing "idiot proof" data entry programs. These are programs which are designed to watch out for every type of mistake that an operator can make when entering data. An idiot proof program will cope with errors in a way that the operator can understand. Chapter 4 describes these data entry programming techniques in detail.

One simple trick worth noting, however, is the INPUT statement's ability to display a short message that can describe the expected input. Such a message is called a *prompt message*. The message appears in the INPUT statement as a string value enclosed in quotation marks. The message will be displayed just ahead of the input request. This certainly beats sticking a bunch of variables into a single INPUT statement, with only your memory reminding you what to enter next.

In Integer BASIC, you put the prompt message immediately following the word INPUT. It is followed by a comma and then the list of variables. When the list contains more than one variable, the prompt message is still only displayed once, on

the first line of input. If the first variable on the list is numeric, a question mark is displayed immediately after the prompt message. If the first variable is a string, no such question mark is displayed. Here is an example:

```
10 DIM A$(10)
20 INPUT "ENTER YOUR NAME AND AGE ",A$,A
30 PRINT A$;"IS";A
35 REM IF ENTRY IS NULL, END PROGRAM
40 IF A$="" THEN END
50 GOTO 20
```

In Applesoft, you put the prompt message immediately after the word INPUT. It is followed by a semicolon which is in turn followed by the variable list. The existence of a prompt message suppresses the standard INPUT statement question mark. The prompt message is displayed only once, even if more than one line is required to enter all of the values requested by the variable list.

Here is an Applesoft example:

```
20 INPUT "ENTER YOUR NAME AND AGE: ";A$,A
30 PRINT A$;" IS ";A
35 REM IF ENTRY IS NULL, END PROGRAM
40 IF A$ = "" THEN END
50 GOTO 20
```

### The GET Statement

The GET statement, available only in Applesoft, inputs a single character from the keyboard. It does not display the character on the screen. You do not press RETURN after typing the character. The entry is treated as a string value or as a numeric value, depending on the type of variable that follows the word GET. Type in the following program and run it:

```
10 GET A$
20 PRINT A$
25 REM IF ENTRY IS AN E, END PROGRAM
30 IF A$ = "E" THEN END
40 GOTO 10
```

We can make GET wait for a specific character by testing for the character as follows:

```
10 GET A$
20 IF A$ < > "X" THEN GOTO 10
30 PRINT A$
40 END
```

This program waits for the letter X to be entered. Nothing else will do.

If the GET statement specifies an integer or real variable, then the entry must be a numeric digit. Otherwise, the ?SYNTAX ERROR occurs and the program stops. Because of this and other problems that can occur when using GET with a numeric variable, GET statements usually receive string characters.

Programs use the GET statement most frequently when generating dialogue

with an operator. For example, a program may wait for an operator to prove that she or he is there by entering a specific character (e.g., Y for yes). Here is appropriate program logic:

```
10 PRINT "OPERATOR! ARE YOU THERE?"
15 PRINT "TYPE Y FOR YES"
20 GET A$
30 IF A$ < > "Y" THEN GOTO 20
40 PRINT "OK, LET'S GET ON WITH IT"
50 END
```

Notice that this sequence never displays the character entered at the keyboard. Try rewriting the program so that any character entered for the GET statement is displayed.

## HALTING AND RESUMING PROGRAM EXECUTION

If a program is running and you want to stop it, press CTRL and C concurrently. If the program is waiting for keyboard input from an INPUT statement when you type CTRL-C, you will also have to press RETURN after you press CTRL-C.

In Integer BASIC, you will see the message STOPPED AT and then the line number at which program execution halted. You can resume program execution by typing the command CON.

In Applesoft, you will see the message BREAK IN followed by the line number at which program execution halted when you pressed CTRL-C. You can continue program execution by typing CONT.

### The RESET Key

You can of course interrupt your program at any time by pressing the RESET key (on some versions of the Apple II, you must press CTRL-RESET).

On the Apple II Plus and on an Apple II with the Language System installed, RESET has the same effect as CTRL-C.

On versions of the Apple II without the Autostart Monitor, pressing RESET causes the Monitor prompt (\*) to appear. If you were using Integer BASIC or Firmware Applesoft, type CTRL-C to return to the language you were using. If you were using cassette-based Applesoft, type OG to return to Applesoft. If you were using disk-based Applesoft, type 3DOG to return to Applesoft.

If you try to recover incorrectly from an accidental RESET, you will lose your BASIC program.

### The END Statement

The program will halt execution when it encounters an END statement, as we described earlier in this chapter.

You cannot continue program execution after Integer BASIC executes an END statement.

### The STOP Statement

Applesoft has another command which will halt program execution when it is executed: the STOP command. When Applesoft executes a STOP command, it displays the message BREAK IN along with the line number it stopped at.

You can continue program execution in Applesoft after either the END or STOP command by typing CONT.

### The WAIT Statement

Applesoft has a command which allows you to specify a pause in program execution. The WAIT statement causes program execution to halt until a memory location (which you specify) has a value which you specify. You can, for example, make your program pause until someone presses the button on game control number 1. Here is how:

```
10 REM -WAIT FOR BUTTON ON GAME CONTROLLER NO.1
20 PRINT "PRESS BUTTON ON GAME CONTROLLER ONE"
30 WAIT - 16286,128
40 PRINT "BANG!"
50 END
```

See Chapter 8 for more complete details on the WAIT statement.

## FUNCTIONS

Another element of BASIC is the function, which in some ways looks like a variable, but in other ways acts more like a BASIC statement.

Perhaps the simplest way of understanding what a function is is to look at an example. In the assignment statement:

```
110 A=SQR(B)
```

the variable A is set equal to the square root of the variable B. SQRT specifies the square root function. Here is a string function:

```
20 L=LEN(D$)
```

In this example the string variable L is set equal to the length of string variable D\$.

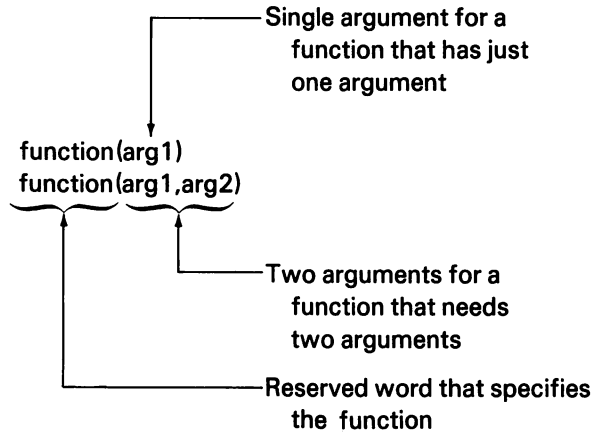
Functions can substitute for variables or constants anywhere in a BASIC statement, except to the left of an equal sign. In other words, you can say that  $A=SQR(B)$ , but you cannot say that  $SQR(A)=B$ .

The discussion which follows shows you how to use functions. An incomplete summary of the functions available in Integer BASIC and Applesoft is presented here but a complete description of all functions is given in Chapter 8. Many functions are not available in Integer BASIC. We will note those that are not.

You specify a function using the appropriate reserved word (such as SQR for square root), followed by an argument or arguments enclosed in parentheses. In

the case of  $A=SQR(B)$ ,  $SQR$  requires a single argument, which in this instance is the value to take the square root of. For  $L=LEN(D\$)$ ,  $LEN$  specifies the function; the argument  $D\$$ , enclosed in parentheses, is the string to take the length of.

Generally stated, any function will have one of these two formats:



A few functions need three arguments. Each function argument can be a constant, a variable, or an expression.

Each and every function in a BASIC statement is reduced to a single numeric or string value before any other parts of the BASIC statement are evaluated. First, the function argument is evaluated according to the rules we set down earlier. Once it is reduced to a numeric or string value, the function is applied to it, yielding another numeric or string value. Not until all functions in a given expression are evaluated in this way is the expression itself evaluated. For example, in the following statement:

```
110 B=24.7*SQR(C)+5)-SIN(0.2+D)
```

$SQR$  and  $SIN$  functions are evaluated first. Suppose  $SQR(C)=6.72$  and  $SIN(0.2+D)=0.625$ . The expression on line 10 will first be reduced to:

$$24.7*(6.72 + 5) - 0.625)$$

then this simpler expression is evaluated.

## NUMERIC FUNCTIONS

Here is a list of the numeric functions that you can use with both Integer BASIC and Applesoft:

- |            |  |
|------------|--|
| <b>SGN</b> | Returns the sign of an argument: +1 for a positive argument, -1 for a negative argument, 0 for a zero argument.                              |
| <b>ABS</b> | Returns the absolute value of an argument. A positive argument does not change; a negative argument is converted to its positive equivalent. |

<b>RND</b>	Generates a random number. See Chapter 8 for more details.
------------	--

Here is a list of the numeric functions that you can use with Applesoft only:

<b>INT</b>	Converts a floating point argument to its integer equivalent.
<b>SQR</b>	Computes the square root of the argument.
<b>EXP</b>	Raises the constant $e$ to the power of the argument ( $e^{arg}$ ).
<b>LOG</b>	Returns the natural logarithm of the argument.
<b>SIN</b>	Returns the trigonometric sine of the argument, which is treated as a radian quantity.
<b>COS</b>	Returns the trigonometric cosine of the argument, which is treated as a radian quantity.
<b>TAN</b>	Returns the trigonometric tangent of the argument, which is treated as a radian quantity.
<b>ATN</b>	Returns the trigonometric arctangent of the argument, which is treated as a radian quantity.

### Using Numeric Functions

You should start using functions as soon as possible, but do not bother with functions you do not already understand. For example, if you do not understand trigonometry, you are unlikely to use SIN, COS and TAN functions in your programs, and there is not much point learning what they are.

Here is an example that uses a numeric function:

```
10 A=-234
20 B= SGN (A)
30 PRINT B
40 END
```

When you execute this program, the result displayed is  $-1$ , since the  $-234$  is negative. As an exercise, change the statement on line 10 to an INPUT. Change line 40 to GOTO 10. Now you can enter a variety of values for A and watch the SGN function at work.

Here is a more complex example using Applesoft BASIC numeric functions:

```
10 INPUT A,B
20 IF LOG (A) < 0 THEN A = 1 / A
30 PRINT SQR (A) * EXP (B)
39 REM USE CTRL-C TO END PROGRAM
40 GOTO 10
```

If you understand logarithms, then as an exercise change the statement on line 20, replacing the LOG function with some other numeric function.

## STRING FUNCTIONS

String functions allow you to manipulate string data in a variety of ways. You may not need to use numeric functions that you do not understand, but you must make an effort to learn every string function.

Here is a list of the string functions that you can use with both Integer BASIC and Applesoft:

ASC	Converts a string character to a standard numeric code (ASCII) equivalent.
LEN	Returns the number of characters contained in a text string.

Here are the string functions you can use in Applesoft only:

STR\$	Converts a numeric value to a string of text characters.
VAL	Converts a string of text characters to their equivalent number (if such a conversion is possible).
CHR\$	Converts a numeric (ASCII) code to its equivalent text character.
LEFT\$	Extracts the left part of a text string. Function arguments identify the string and the desired left part.
RIGHT\$	Extracts the right part of a text string. Function arguments identify the string and the desired right part.
MID\$	Extracts the middle section of a text string. Function arguments identify the string and the required mid part.

String functions let you determine the length of a string, extract portions of a string, and convert numeric values, numeric (ASCII) codes, and string characters. These functions take one, two, or three arguments. Here are some examples:

STR\$(14)	Converts 14 to "14".
LEN("ABC")	Returns the length of the string. The number 3 is returned since the string has three characters.
LEN(A\$+B\$)	Returns the combined length of the two strings.
LEFT\$(ST\$,1)	Returns the leftmost character of the string ST\$.

## Integer BASIC Substrings

Although Integer BASIC has no functions that let you extract portions of a string, there is a way of doing it. You specify the starting position and the number of characters in the substring, as in the following example:

```
10 DIM A$(20),B$(5)
20 B$=A$(1,4)
```



In the example above, B\$ is set equal to the first four characters of A\$. It may look to you as though B\$ is being assigned the value of one of the elements of string array A\$( ), but remember that Integer BASIC does not allow string arrays, much less double-dimension string arrays. Instead, this notation refers to a substring. The first value in parentheses is the starting position of the substring, and the second value is the number of characters in the substring.

### Integer BASIC String Concatenation

The LEN function allows you to concatenate strings in Integer BASIC. Here is an example:

```

10 DIM A$(10),B$(10),C$(10)
20 A$="WIND"
30 B$="PIPE"
40 C$="LINE"
50 A$( LEN(A$)+1)=B$
60 PRINT A$
70 B$( LEN(B$)+1)=C$
80 PRINT B$
90 END

>RUN
WINDPIPE
PIPELINE

```

### SYSTEM FUNCTIONS

In the interest of completeness, system functions are listed below. They perform operations which you are unlikely to need until you are an experienced programmer.

Here is a list of system functions available:

PEEK	Fetches the contents of a memory location.
FRE	Returns available free space — the number of unused read/write memory bytes. Not available in Integer BASIC.
USR	Transfers to an assembly language program. Not available in Integer BASIC.

### USER-DEFINED FUNCTIONS

In addition to the many functions which are a standard part of BASIC, you can define your own arithmetic functions in Applesoft, providing they are not very complicated. User-defined string functions are not allowed. A DEF FN statement defines the function. You invoke the function with an FN statement. Here is a short program that uses a DEF FN statement:

```

10 DEF FN P(X) = 100 * X
20 INPUT A

```

```
30 PRINT A, FN P(A)
35 REM USE CTRL-C TO END PROGRAM
40 GOTO 20
```

The function identifier follows the reserved word FN. Rules for naming the identifier are the same as rules for real variable names. In the example we use P, therefore the function name becomes FNP. If the identifier were AB, then the function name would be FNAB.

The arithmetic expression on the righthand side of the equal sign defines the function. When you invoke the function with the FN statement, the expression is evaluated (using the current values for any variables mentioned in the expression). The resulting numeric value is treated the same as any numeric value would be in the context where the FN statement appears.

In the DEF FN statement, a single variable must follow the function identifier, and must be enclosed in parentheses. This variable name is *local* to the function definition; it has no effect outside of the DEF FN statement. You can use the same variable name elsewhere in the program without affecting the function, and the variable in the function will not affect any like-named variable elsewhere in the program.

In use, the FN statement must be followed by the function identifier, which must be followed by a numeric constant, variable, or expression enclosed in parentheses. When Applesoft encounters the FN statement, it assigns the value of the constant, variable, or expression to the local variable in the DEF FN statement. (The value of a variable outside the DEF FN statement which has the same name as the local variable is unchanged.) If the local variable appears in the function-defining expression, Applesoft uses its recently assigned value when evaluating the expression.

## FUNCTION NESTING

The argument of a function can be an expression; the expression may contain functions. In other words, functions can be nested. Here is an example:

```
10 INPUT A
20 PRINT SGN ( ABS (A) )
25 REM USE CTRL-C TO END THE PROGRAM
30 GOTO 10
40 END
```

Try experimenting by creating immediate mode PRINT statements that make complex use of numeric and string functions.

# 4

## Advanced BASIC Programming

This chapter carries on from Chapter 3 in describing how to program the Apple II in BASIC. It covers many new BASIC statements and explores new facets of some familiar ones. Chapter 3 taught you enough to let you make your Apple II do some fancy tricks; this chapter shows you how to make it a really useful tool.

### DIRECT ACCESS AND CONTROL

There are a number of statements which allow you direct access to the Apple II. There are many things you can do only via these statements — things like sensing the game controls, operating the speaker, and making full use of peripheral devices.

### MEMORY AND ADDRESSING

The Apple II can have up to 65,536 individually addressable memory locations, each of which can store a number ranging between 0 and 255. (This strange upper bound is in fact  $2^8$ .) All programs and data are converted into sequences of numbers which are stored in this fashion.

You must specify a memory location for each of the following BASIC statements. You can specify the address with a number, a variable, or an expression. In

any case, it must evaluate to a valid memory location. There are two valid addresses for each memory location. One is positive and is an integer between 0 and 65535. The other is negative and can be derived by subtracting 65536 from the positive address. For example, -32767 and 32768 address the same memory location. Another memory location is addressed by either -1 or 65535.

When you remember that the largest number allowed in Integer BASIC is 32767, you can see the utility of using negative numbers for addressing the higher memory locations.

If you specify a memory location in Applesoft with a real value, it will be converted to an integer value for use as an address.

### **PEEK and POKE**

The PEEK function lets you read the value stored in any Apple II memory location. Consider the following statement:

```
10 A = PEEK(200)
```

This statement assigns the content of memory location 200 to variable A.

The POKE statement puts a value into a memory location. For example the statement:

```
20 POKE 8000,A
```

takes the value of variable A and stores it in memory location 8000. The value to be written into memory may be a number, a variable, or an expression with a value between 0 and 255.

You can PEEK into read/write memory or read-only memory (RAM or ROM). But you can POKE only into read/write memory. This is self-evident; read-only memory, as its name implies, can have its contents read, but cannot be written into.

### **CALL Statement**

You can transfer control from BASIC to an assembly language program or subroutine with the CALL instruction. Look at this sample CALL statement:

```
100 CALL A1
```

Control transfers to the memory location specified by variable A1.

The assembly language subroutine or program can be one that is constantly resident in the Apple II (in ROM) or it can be one you provide. The Monitor has an intrinsic subroutine that will clear the display screen, for example. Appendix D has a complete list of intrinsic subroutines you may find useful. Also, refer to Chapter 7 for more coverage of the Monitor and assembly language.

### **HIMEM: and LOMEM: Statements**

The memory in your Apple II is used in different ways. Part of it is used to store your BASIC program, part for your program variables, another part for programs

to manage disk drives (if you are using disk drives), and so on. Some of the memory is used for graphics, as we will learn later in Chapter 6.

The statements HIMEM: and LOMEM: allow you to set aside safe areas of memory for assembly language programs and high-resolution graphics.

Here is an example of each of these statements:

```
50 HIMEM: 38400
60 LOMEM: 12291
```

The upper and lower boundaries are set automatically by the Apple II to the highest and lowest memory locations available for a BASIC program on your particular Apple II. You may need to reset these boundaries if you are using high-resolution graphics or assembly language programs, which we will cover in Chapters 6 and 7. For more information on memory uses, see Appendix G.

## USING PERIPHERAL DEVICES

When you turn the Apple II on, it expects input from the keyboard and displays its output on the screen. The keyboard is the standard input device and the display screen is the standard output device. But as you are no doubt aware, the Apple II can communicate with peripheral input and output devices. These include:

- A cassette recorder for loading and saving programs (and some data)
- Disk drives for loading and saving programs and data
- A printer for permanent paper copies of data and programs
- A graphics tablet for inputting freehand graphic designs
- Communications with other computers.

The cassette recorder plugs into the Apple II through two special jacks.

All other input and output devices plug into add-on electronic circuit boards which connect to slots inside the main Apple II computer. These circuit boards are called *controllers*, *interface cards*, *cards*, or *interfaces*. You specify which device you want to use by the slot number its controller is connected to. There are eight slots, numbered 0 through 7. The screen display and keyboard are always slot 0.

### The PR# and IN# Statements

To select a different output slot use the PR# statement. For a different input source use IN#. The following immediate mode example selects the screen display for output:

```
PR#0
```

As long as the Disk Operating System (DOS) is not present, you use PR# and IN# the same way in programmed mode and immediate mode. Using PR# and IN# in programmed mode with DOS present is a bit more complicated. You must use a PRINT statement which prints a CTRL-D character followed immediately by

the PR# or IN# command. The following statements direct subsequent PRINT statement output to slot 1. Via the interface in this slot, output will go to a printer or any other device connected to the card.

```
100 D$ = "": REM CTRL-D
110 PRINT D$;"PR#1": REM SELECT SLOT NUMBER 1 (PRINTER)
```

With DOS absent, you would use the following statement instead:

```
110 PR#1 : REM SELECT SLOT 1 (PRINTER)
```

Notice that PR# and IN# each have a single parameter. It must be a numeric value between 0 and 7. Any other value will cause unpredictable results. If there is no interface card in the slot selected by PR# or IN#, then the Apple II locks up; your only recourse is to press RESET.

## PROGRAM OUTPUT AND DATA ENTRY

The most inexperienced programmer quickly discovers that the input and output sections of a program are its trickiest parts.

Nearly every program uses data which must be entered at the keyboard. Will a few INPUT statements suffice? In most cases the answer is no. What if the operator accidentally presses the wrong key? Or worse, what if the operator discovers that he or she input the wrong data — after entering two or three additional data items? A usable program must assume that the operator is human, and is likely to make any conceivable human error.

Results, likewise, cannot simply be displayed, or printed, by executing a bunch of PRINT statements. A human being will have to read this output. Unless the output is carefully designed, it will be very difficult to read; in consequence information could be misread, or entirely overlooked.

Fortunately BASIC on the Apple II has many capabilities that make it easy to program input and output correctly. We will describe some of these capabilities before looking specifically at good input and output programming practices.

### MORE ABOUT THE PRINT STATEMENT

Normally a PRINT statement ends by returning to the beginning of the next display line (it performs a *carriage return*). This causes the next PRINT statement to begin displaying in the first character position of the next line. Thus the following program displays a column of 20 W characters in the first character position of 20 rows:

```
>NEW
>200 C$="W"
>210 FOR I=1 TO 20
>220 PRINT C$
>230 NEXT I
```

```
>240 PRINT "PHEW!"
>250 END
>RUN
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
W
PHEW!
```

## Using Semicolons

A semicolon (;) appearing after any variable in the PRINT statement parameter list causes the next display to begin immediately at the next available character position. A semicolon following the last (or only) variable in the PRINT statement suppresses the carriage return. Therefore the following program will display 800 W characters across 20 rows of a 40-column display.

```
>NEW  

>200 C$="W"  

>210 FOR I=1 TO 800  

>220 PRINT C$;  

>230 NEXT I  

>240 PRINT "PHEW!"  

>250 END  

>RUN
```









[illegible]

Numbers are broken across the end of lines if necessary. This is because the semicolon (;) generates a continuous display, and nothing but an end of line can cause a return. Try the example above with C equal to 201 to see this.

## Using Commas

Commas appearing after a variable or at the end of a PRINT statement treat the display as though it were tabbed at specific intervals. The rules for tabbing are different in Integer BASIC and Applesoft.

Integer BASIC establishes five tab stops on the display screen. They are at columns 1, 9, 17, 25, and 33; they are eight spaces apart. Here is an Integer BASIC example:

[illegible]

```

123      123      123      123      123
123      123      123      123      123
123      123      123      123      123
PHEW!

```

There must be at least one blank space ahead of a tab stop (except the first tab stop) or that tab stop will be inactivated. Thus, there must not be anything printed in column 8 for the second tab stop to be active, and so on. The following example program illustrates this with strings.

```

>10 REM MUST DIMENSION STRING VARIABLES
>20 DIM C$(8)
>200 C$="12345678"
>210 FOR I=1 TO 20
>220 PRINT C$,
>230 NEXT I
>240 PRINT "PHEW!"
>250 END
RUN
12345678      12345678      12345678
      12345678      12345678
12345678      12345678      12345678
      12345678      12345678
12345678      12345678      12345678
      12345678      12345678
12345678      12345678      12345678
      12345678      12345678
PHEW!

```

If you change C\$ to "1234567," all five tab stops will be used.

There are some special conditions which govern tabbing with commas in Applesoft, as shown in Figure 4-1.

To see how commas work in Applesoft using our example program, change the semicolon in the PRINT statement to a comma. This causes numbers to be displayed in three columns. At three numbers per line, the FOR-NEXT index will be  $3 * 20 = 60$ . The complete program is shown at the top of the next page.

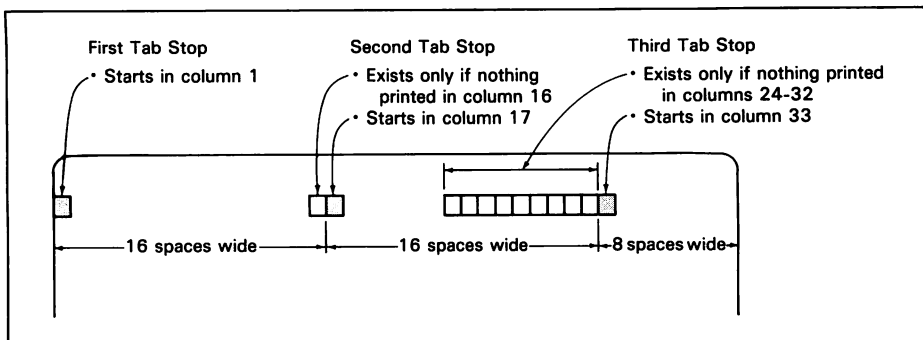


FIGURE 4-1. Applesoft Tab Stops From PRINT With Commas

[illegible]

Commas also work with strings. As an example, enter the following immediate mode program to display twenty lines of tabbed character data:

```
JNEW
J100 A$="THREE"
J110 B$="BLIND"
J120 C$="MICE"
J210 FOR I=1 TO 20
J220 PRINT A$,B$,C$
J230 NEXT I
J240 PRINT "PHEW!"
J250 END

JRUN
THREE          BLIND          MICE
THREE          BLIND          MICE
```



The heading is approximately centered on the display screen. Were the heading longer, say REPORT ON HEAVY HYDROGEN, it would not be centered on the display screen. (Try it.) This new heading will be approximately centered if you remove one of the commas and its adjoining null string, so that the heading is displayed starting at the second tab stop rather than the third.

We could use the same technique for displaying headings in Applesoft. But Applesoft has PRINT statement formatting aids that are much easier to use: the SPC, TAB, and POS functions.

### SPC Function

SPC is a space-over function. You include SPC as one of the terms in a PRINT statement; after the letters SPC you must enclose (in parentheses) the number of character positions that you wish to space over. For example, we could center the heading TOP SECRET with the following:

```

1100 REM CLEAR SCREEN

1110 HOME

1120 REM SPACE OVER AND DISPLAY

1130 PRINT SPC(15);"TOP SECRET"

1140 END

1RUN
                                TOP SECRET

```

Notice the semicolon after SPC. A comma after SPC will start displaying text at the next tab stop following the number of spaces specified by SPC.

Any time you include the SPC function in a PRINT statement you simply cause the next printed or displayed character to be moved over by the number of positions specified after SPC; nothing else is changed.

### TAB Function

TAB works much the same as typewriter tabbing.

Suppose you want to print or display information in columns. You must first calculate the character position where each column is to begin. A form like the one in Appendix L is handy for this. In Figure 4-2, columns begin at character positions 1, 15, and 31. Now instead of using blank spaces as you go from column to column, you simply insert a TAB function after each entry in the PRINT statement.

Consider one line of the display illustrated in Figure 4-2. Counting character positions, we could display the line without tab stops, as follows:

```

110 ?"JONES,P.J          431-25-6277      1420.00"

```





### **Determining Cursor Position (Vertical)**

PEEK(37) will give you the row which the cursor is currently on.

Rows are numbered 0 through 23 for PEEK(37), but 1 through 24 for VTAB (covered in the next section).

### **CURSOR CONTROL AND SPECIAL VIDEO EFFECTS**

There are a number of BASIC statements which increase the versatility of the Apple II display screen. These include the FLASH, INVERSE, SPEED, NORMAL, HOME, VTAB, and HTAB/TAB statements. Most of these are available only with Applesoft. There are a number of graphics statements as well; these are covered in Chapter 6.

#### **Positioning the Cursor**

We have already covered several ways of controlling the cursor position by using commas and semicolons with the PRINT statement. The Applesoft formatting functions SPC, TAB, and POS are also useful in positioning the cursor.

#### **Clearing the Display Screen**

You can clear the display screen and position the cursor to the home position (upper left corner) with the statement CALL -936, or in Applesoft with the HOME statement. Try typing these (one in Integer BASIC, the other in Applesoft):

```
>CALL -936
```

```
]HOME
```

#### **Horizontal and Vertical Positioning**

There are two statements which together allow you to move the cursor to any position on the screen. VTAB moves the cursor vertically and HTAB (TAB in Integer BASIC) moves the cursor horizontally. You must specify the line number for VTAB and the column number for HTAB. The top line of the screen is 1 and the bottom line is 24. The leftmost column of the screen is 1 and the rightmost column is 40.

The following program uses these two statements (in Applesoft) to position the cursor and display an asterisk at that position. If you are using Integer BASIC use TAB instead of HTAB on lines 90 and 120.

```

90 HTAB 1: VTAB (1)
100 INPUT "ROW?";R
110 INPUT "COLUMN?";C
120 VTAB R: HTAB C
130 PRINT "*";
140 GOTO 90

```

### The INVERSE and NORMAL Statements

You can reverse the black and white parts of characters on the display screen; the INVERSE statement does this. Once the INVERSE statement is executed, everything displayed by PRINT statements appears in *reverse video* mode. The characters you type on the keyboard will still echo in normal video mode, however.

The Apple II returns to normal video mode when it executes a NORMAL command.

To see how these two commands work, try this (the shaded characters will be in reverse video):

```

]INVERSE

]?"BLACK ON WHITE"
BLACK ON WHITE

]NORMAL

]?"WHITE ON BLACK"
WHITE ON BLACK

```

INVERSE and NORMAL are not available in Integer BASIC.

### The FLASH Statement

Not only can you reverse the black and white parts of characters, you can make characters flash back and forth between normal and reverse video. Use the FLASH statement to do this. Here again, NORMAL causes the Apple II to revert to normal video.

Here is an example (the shaded areas will be flashing):

```

]FLASH.

]?"FLASH IN THE PAN"
FLASH IN THE PAN

]NORMAL

]?"STEADY AS RAIN"
STEADY AS RAIN

```

FLASH is not available in Integer BASIC.

### The SPEED Statement

The rate at which characters display on the screen is variable. You can slow it down from its normal rate with the SPEED statement.

The following program illustrates how **SPEED** works.

```

1100 INPUT "SPEED = ";SP
1110 SPEED = SP
1120 FOR CT = 1 TO 3
1130 PRINT "HIC"
1140 NEXT
1150 PRINT "HICCUP"
1160 SPEED = 255
1170 END

```

The value of the expression (e.g., the value of **SP** on line 110) adjusts the display speed; 0 is slowest and 255 is fastest. **SPEED** also affects the rate at which characters are sent to other devices, not just the display screen.

**SPEED** is not available in Integer BASIC.

## TEXT WINDOWS

Normally, the Apple II actively uses all 24 lines and 40 columns of its display screen in text mode. Using **POKE** statements, you can alter the size and position of this *text window*. Four memory locations control the size, shape, and position of the text window, as shown in Table 4-1.

You must be careful to observe the common-sense range limits shown in Table 4-1 when you set a text window. Trying to set windows outside these bounds will have unpredictable results.

The sample program below sets a two-line text window in the middle of the display screen for inputting a numeric value. To appreciate the full utility of this technique, try entering some non-numeric values and observe what happens with the error messages. Notice also that setting the text window does not clear the remainder of the screen, nor does it move the cursor into the text window. You must provide separate BASIC statements to do this.

TABLE 4-1. Text Window Memory Locations

Memory Location	Controls	Allowable Range
32	Left Margin	0 to 39
33	Width	1 to 40 minus Left Margin
34	Top Line	0 to Bottom Line
35	Bottom Line	Top Line minus 24

```
1000 REM SET WINDOW TOP LINE, WIDTH, LEFT MARG., & BOTTOM
    LINE
1010 T = 10:W = 20:LM = 11:B = 13
1020 REM CLEAR SCREEN
1030 CALL - 936
1040 REM SET TEXT WINDOW FOR INPUT
1050 GOSUB 3200
1060 REM SURROUND INPUT WINDOW WITH ASTERISKS
1070 GOSUB 3000
1080 REM MOVE CURSOR INSIDE WINDOW: INPUT
1090 VTAB T + 2
1100 INPUT M1
1110 REM RESET WINDOW TO FULL SCREEN
1120 GOSUB 3300
1130 REM MOVE CURSOR TO BOTTOM LINE
1140 VTAB 23
1150 END
2990 REM SURROUND WINDOW WITH ASTERISKS
3000 VTAB T + 1
3010 GOSUB 3100
3020 VTAB B + 1
3030 GOSUB 3100
3040 RETURN
3090 REM PRINT ASTERISKS
3100 FOR I = 1 TO W
3110 PRINT "*";
3120 NEXT I
3130 RETURN
3190 REM SET INPUT TEXT WINDOW
3200 POKE 32,T
3210 POKE 33,W
3220 POKE 34,LM
3230 POKE 35,B
3240 RETURN
3290 REM SET FULL-SCREEN WINDOW
3300 POKE 32,0
3310 POKE 33,40
3320 POKE 34,0
3330 POKE 35,24
3340 RETURN
```

## THE CHR\$ FUNCTION: PROGRAMMING CHARACTERS IN ASCII

In the last chapter we discussed how you can generate invisible characters like CTRL-G, which makes the Apple II beep. But there are some characters (both visible and invisible) which you cannot type directly on the keyboard. These include "[" and "\". You can generate these characters in Applesoft with the CHR\$ function.

In order to understand the CHR\$ function, you must understand how characters are stored in the Apple II memory. It's really very simple. Computer memory can store numbers, but not characters. So characters are converted to numeric

codes. The Apple II uses the same code as all other microcomputers, the ASCII code (American Standard Code for Information Interchange). For example, the ASCII code for the letter A is 65, for B it is 66, C is 67, and so on. You will find a complete table of ASCII codes in Appendix I. Whenever the Apple II is dealing with strings, it interprets numeric values as ASCII codes for characters.

In Applesoft, if you cannot press a key to include a character within a text string, you can still select the character by using its ASCII code.

The CHR\$ function translates an ASCII code number into its character equivalent. For example, to create the symbol "\$", first find its ASCII code in Appendix I. Then use the code with CHR\$ as follows:

```
JPRINT CHR$(36)
$
```

Experiment in immediate mode using any ASCII code from 0 to 255.

You can use the CHR\$ function in conjunction with regular strings in a PRINT statement as follows:

```
J?CHR$(34); "ZOUNDS!"; CHR$(34)
"ZOUNDS!"
```

The CHR\$ function lets you include otherwise unavailable characters such as carriage return and quotation marks as part of a string.

## PROGRAMMING DATA ENTRY

The goal of any program should be to minimize data entry errors and make it easy for an operator to spot and correct errors that do occur. There are ways to organize data entry which tend to minimize errors.

First, enter a whole functional block of data and *then* process it. It is poor programming practice to process each piece of data as it comes in from the keyboard.

A mailing list program, for example, requires names and addresses to be entered as data. You should treat each name-and-address as a single functional unit. In other words, your program should ask for the name and address, allowing the operator to enter all of this information, and then change any part of it. When the operator is satisfied that the name and address is correct, the program can go ahead and process it. Then the program should ask for the next name and address.

In the case of a mailing list program it would be poor programming practice to ask for the name, process it immediately, then ask for each line of the address, treating each piece of the name and address as a separate and distinct functional unit.

It is a good idea to organize data entries so that data items remain on the display screen for a while after they have been entered. That gives the operator more opportunity to notice an error and correct it. If entries disappear as soon as they are entered, there is no chance for an operator to casually notice an error. Of course, the operator must have some way of correcting previous entries.

Under some circumstances entering data in functional blocks with ample opportunity for review and correction is not the best way to go. This set of circumstances is a surprising one. It occurs when a very large amount of data must be entered by keyboard operators. For example, suppose a keyboard operator must enter hundreds of names and addresses a day. Experience has shown that the highest volume of accurate data entry can be achieved by having the keyboard operator ignore all errors on first entry. The data entry program does not allow for the correction of any errors, even if the operator notices them right away. Operators must ignore errors and carry on entering data as fast as possible. Under this scheme, data is entered twice, preferably by different operators. A separate program compares the two sets of entries. The chances of both operators making the same error are so small that you can count on all errors being caught as differences between the two sets of data entry. A subsequent program allows reentry of incorrect data.

### Interactive Data Entry

A program with interactive data entry guides users with displayed instructions and prompt messages. To demonstrate interactive data input we will begin with a very simple example. We will make changes to a program you will remember from earlier in the chapter so that it uses interactive data entry procedures, thereby making the program easier to use.

Here is the program:

```
200 C$ = "W"
210 FOR I = 1 TO 800
220 PRINT C$;
230 NEXT I
240 PRINT "PHEW!"
250 END
```

As illustrated, this program will display 800 W characters followed by the word PHEW!. It works the same in Integer BASIC or Applesoft.

Suppose we want to display any character, instead of just W.

First eliminate the assignment statement in the program. Remember that to delete a program statement, you type the line number followed immediately by a RETURN.

```
210 FOR I = 1 TO 800
220 PRINT C$;
230 NEXT I
240 PRINT "PHEW!"
250 END
```

Line 200 is no longer in the program.

Type in the statement C\$="X" in immediate mode, then run the program.

```
]C$="X"

]RUN
PHEW!
```



Run the program. The cursor drops down one line. (In Applesoft a question mark appears next to the cursor, too.) Enter any *single* character (and press RETURN). The character you enter is displayed 800 times. Run the program again. Enter a different character. The display appears with the new character. Note that if you try to enter more than one character at a time in Integer BASIC, the program halts with a \*\*\* STR OVFL ERR. This is because C\$ has not been dimensioned.

This is a real improvement over the original program. However, it is a little mystifying to have the cursor just blinking away on the screen while it waits for you to press a key. Add a prompt line to the beginning of the program, asking for a key to be pressed. Type in the line:

```
1190 PRINT "ENTER ONE CHARACTER"
```

List the program and check the new line for any errors.

Now the program gives operating instructions. Run the program several times to display different characters and note how much easier the program is to use.

There is one important modification left to make. If you want the program to repeat automatically, use a GOTO statement to go back to the beginning of the program instead of ending it. Then you won't have to type in RUN to reexecute the program. Add the following line:

```
1250 GOTO 190
```

Now it is even easier to use the program. Enter RUN and follow directions.

Of course, you have to use CTRL-C to end the program. This can be eliminated by programming one particular key to terminate program execution. For example, the RETURN key could be programmed to terminate execution. Here is how:

```
190 PRINT "ENTER ONE CHARACTER"
200 INPUT C$
205 IF C$ = "" THEN END
210 FOR I = 1 TO 800
220 PRINT C$;
230 NEXT I
240 PRINT "PHEW!"
250 GOTO 190
```

Line 205 checks to see if C\$ has a null value. This will happen if you do not enter a character in response to the INPUT statement on line 200, but only press the RETURN key.

As a final task, you might read over the program and add remarks. Include a comment on how the FOR-NEXT index 800 was devised; you can optionally put the remark on the same line, using a colon to separate statements:

```
210 FOR I=1 TO 800:REM 800/40=20 LINES
```

Add a reminder that a null entry ends the program:

```
203 REM END PROGRAM ON NULL ENTRY
```

Finally, add a few lines at the beginning of the program to describe it, and give it a title. The final program is shown in Figure 4-3.



```

10 REM ***** BLANKET *****
20 REM CONTINUOUS-LINE DISPLAY OF ONE
30 REM CHARACTER ENTERED FROM
40 REM THE KEYBOARD
50 REM *****
190 PRINT "ENTER ONE CHARACTER"
200 INPUT C$
203 REM END PROGRAM ON NULL ENTRY
205 IF C$ = " " THEN END
210 FOR I = 1 TO 800: REM 800/40 = 20 LINES
220 PRINT C$;
230 NEXT I
240 PRINT "PHEWI"
250 GOTO 190

```

FIGURE 4-3. Program BLANKET

### Prompting Messages

Any program that requires data entry should prompt the operator by asking questions. Questions are usually displayed on a single line and demand a simple response such as "yes" or "no." For example, the following message might be displayed:

DO YOU WANT TO MAKE ANY CHANGES?

An operator must respond to this message by entering the word YES or the word NO. Frequently just the letter Y or N suffices. Another common example may give the operator a number of options. The message:

WHICH ENTRY DO YOU WISH TO CHANGE?

may allow the operator to enter a code which identifies an allowed entry.

Programs that control this type of dialogue should be written as independent subroutines which do not make any assumptions nor depend on knowledge of programs which call them. This has three implications:

1. You cannot assume that the space where the prompt message will be displayed is blank. If it is not blank, then the message will overwrite whatever was previously there. But existing characters not overwritten by the prompt message will still be there. This could be unsightly from the operator's viewpoint. It can be confusing and lead to entry errors. The following subroutine will erase the number of spaces determined by the value of variable ER:

```

5000 REM ERASE SPACES
5010 FOR I = 1 TO ER: PRINT " ";: NEXT I: RETURN

```

2. A subroutine must receive information from the calling program. For example, if a subroutine asks the operator to enter a number, then any program that uses it must specify the minimum and maximum acceptable entry value.
3. The subroutine must return the operator's response to the calling program. This may be a character (e.g., Y or N), a word (e.g., YES or NO) or a number.

Subroutine logic cannot deduce where on the screen it should display the prompt message, however. It is therefore fair to demand that the calling program position the cursor correctly before it calls the subroutine.

Now look at the subroutine needed to ask a question that requires a reply of Y for yes, or N for no. We will use a PRINT statement to ask the question, followed by an INPUT statement to receive a one-character response. We will clear parts of the display screen with the subroutine above. Here is a program with the subroutines:

```

100 REM MOVE CURSOR
140 VTAB 1
150 REM CALL SUBROUTINE TO GET RESPONSE
160 GOSUB 3020
170 END
3000 REM ++ GET Y/N RESPONSE ++
3010 REM
3020 C = POS (0): REM REMEMBER CURSOR COLUMN
3030 R = PEEK (37): REM REMEMBER CURSOR ROW
3040 REM CLEAR SPACE FOR QUERY
3050 ER = 35: REM CLEAR 35 SPACES
3060 GOSUB 5010
3070 HTAB C + 1: REM REPOSITION CURSOR
3080 PRINT "DO YOU WANT TO MAKE ANY CHANGES";
3090 INPUT R$
3100 IF R$ = "Y" OR R$ = "N" THEN RETURN
3110 REM IMPROPER RESPONSE
3120 VTAB R + 1: REM REPOSITION CURSOR (VERT)
3130 GOTO 3050: REM TRY AGAIN
5000 REM ++ ERASE SPACES ++
5010 FOR I = 1 TO ER: PRINT " ";: NEXT I: RETURN

```

Next consider dialogue which allows an operator to enter a number. We will design a subroutine to accept a response no smaller than the value of variable LO and no larger than the value of variable HI. The subroutine will return the entered number in variable NM. Here is the necessary program:

```

100 REM SET RANGE AND POSITION CURSOR
140 LO = 1:HI = 10
150 VTAB 1
160 REM CALL SUBROUTINE TO GET RESPONSE
170 GOSUB 3500
180 END
3500 REM ++ GET NUMERIC RESPONSE ++
3510 REM ++ LO<=RESPONSE<=HI++

```

```

3520 REM ++ NM IS RESPONSE ++
3530 GOSUB 5110: REM WHERE IS CURSOR NOW?
3540 REM CLEAR SPACE FOR QUERY
3550 ER = 35: REM CLEAR 35 SPACES
3560 GOSUB 5010
3570 HTAB C + 1: REM REPOSITION CURSOR
3580 PRINT "WHICH FIELD DO YOU WANT TO CHANGE (1-10)";
3590 INPUT NM
3600 IF NM > = LO AND NM < = HI THEN RETURN
3610 REM IMPROPER RESPONSE
3620 VTAB R + 1: REM REPOSITION CURSOR (VERT)
3630 GOTO 3550: REM TRY AGAIN
5000 REM ++ ERASE SPACES ++
5010 FOR I = 1 TO ER: PRINT " ";: NEXT I: RETURN
5100 REM ++ REMEMBER CURRENT CURSOR POSITION ++
5110 C = PEEK (36): REM COLUMN
5120 R = PEEK (37): REM ROW
5130 RETURN

```

Can you change the subroutine so that it accepts two-digit inputs? Try to write this modified program for yourself. If you cannot do it, then wait until later in the chapter where you will find the necessary subroutine in the program which controls the input of a date.

There is another simple modification you can make to both of the dialogues we have described. The prompt message printed in both programs could be supplied by the calling program via a string variable. This would make the subroutines more general purpose. Can you rewrite the programs to accept a message provided by the calling program?

## Error Detection and Control

If you want to write a really thorough program you will make every effort to anticipate errors that a user of your program might make. Your program will catch entry errors and force the user to reenter values that would cause the program to halt abnormally (termed *bombing*, *bombing out*, or *blowing up* in computer jargon).

It is true that the Apple II will catch some kinds of data entry errors for you. It will not accept alphabetic entry when inputting a numeric value with a statement like INPUT A. If you try to enter letters in response to such a statement, the Apple II issues an error message and asks you to reenter the value.

Built-in error checking capabilities are limited, though. It is very possible to enter the right kind of value (e.g., numeric or string) which has an unacceptable value. That is, the value may cause a program error further down the line. Here is a short program that illustrates the problem:

```

100 INPUT X
200 PRINT 100 / X
300 END

```

If you enter 0 in response to the INPUT statement, the program will fail when it tries to divide by 0 in the PRINT statement. It's easy enough to avoid this. The following lines will check the input to make sure it is not 0, and will request reentry if it is.

```
110 IF X < > 0 THEN 200
120 PRINT "NOT ALLOWED...RE-ENTER"
130 GOTO 100
```

By extending the principle illustrated in this example, you can see how easy it is to check an entered value for the correct range. Depending on the circumstances, it may make sense to do range checking with the Applesoft ON-GOTO or ON-GOSUB statements (computed GOTO or GOSUB in Integer BASIC), rather than a series of IF-THEN statements. There is an example of more extensive error checking in the next section.

### **The ONERR GOTO and RESUME Statements**

Applesoft has a special statement that allows you to trap errors which it catches — before it displays an error message and halts program execution. Here is an example:

```
50 ONERR GOTO 8000
```

Once such a statement has been executed, Applesoft will branch to the specified line number if it detects an error. It will also place a numeric code describing the error in memory location 222, which you may inspect with the PEEK statement. Table C-1 in Appendix C lists error conditions detectable by ONERR GOTO.

The usual procedure for handling errors with ONERR GOTO is to write an error handling routine which the ONERR GOTO branches to when an error occurs. At the end of this routine, the RESUME statement causes a branch back to the beginning of the statement where the error occurred. Alternately, a GOTO statement will transfer to any program line. Write the error handling routine so it takes different actions depending on the nature of the error and the current state of the program, which can usually be determined by inspecting the values of key variables.

To negate the ONERR GOTO and restore the Apple II to its normal automatic error handling state, use the statement POKE 216,0.

The following program demonstrates the use of ONERR GOTO. In this program, any errors that cannot occur as the result of keyboard entries are treated as fatal errors, with an appropriate advisory message. Entry errors are announced and reentry requested.

```

50  ONERR GOTO 8000
200  PRINT "ENTER A STRING VALUE"
210  INPUT X$
220  PRINT "ENTER A NUMERIC VALUE"
230  INPUT X
240  PRINT "ENTER AN INTEGER VALUE"
250  INPUT X%
260  GOTO 200
500  REM  END OF PROGRAM
510  PRINT "LAST ENTRIES WERE ";X$;", ";X;" AND ";X%
515  POKE 216,0: REM  TURN OFF ON ERR
520  END
8000  REM  ++ ERROR HANDLING ROUTINE ++
8010  E = PEEK (222): REM  GET ERROR NO.
8020  IF E = 255 THEN GOTO 500: REM  END PROGRAM ON CTRL-C
8030  IF E = 53 OR E = 176 OR E = 254 THEN 8100
8035  INVERSE
8040  REM  PROGRAMMING ERROR DETECTED
8050  PRINT "ARRGH! ERROR NO. ";E;" FOUND."
8055  PRINT "WRITE DOWN THIS NUMBER"
8060  PRINT "AND A DESCRIPTION OF WHAT WAS GOING ON."
8070  PRINT "CALL A PROGRAMMER FOR HELP."
8080  PRINT "LEAVE THE COMPUTER ON!"
8090  NORMAL : STOP
8100  REM  INPUT ERROR DETECTED
8110  PRINT "": REM  CTRL-G CHARACTERS IN QUOTES
8130  PRINT "ERROR...TRY AGAIN"
8140  RESUME

```

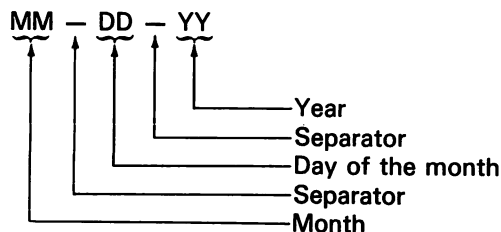
## Entering a Valid Date

In this section we will develop a program that uses many of the techniques presented so far in this chapter. This program uses some BASIC functions and statements available only in Applesoft. The program can be written in Integer BASIC; you may wish to make the conversion.

Most programs at some point need relatively simple data input: more than a simple yes or no, but less than a full screen display. Consider a date.

You must take more care with such simple data entry than might at first appear necessary. In all probability the date will be just one item in a data entry sequence. By carefully designing data entry for each small item, you can avoid having to restart or back up in a long data entry sequence whenever the operator messes up a single entry.

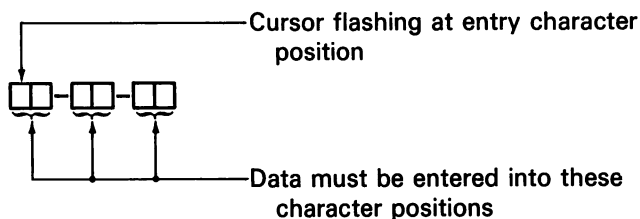
**We will assume that the date is to be entered as follows:**



The month, day of the month, and year are each entered as a two-digit number, without any terminating RETURN.

The program supplies the dash that separates the entries. Depending on your personal preferences, it might be a slash or any other visually pleasing character. In many parts of the world the day of the month precedes the month.

Program data entry so that it is pleasing to the operator's eye. The operator should be able to see immediately where data is to be entered, what type of data is required, and how far the data entry process has proceeded. A good way of showing where data is to be entered is to reverse the data entry field. For example, the program that asks for a date to be entered might create the following reverse field display:



You can create such a display with the following program:

```

10 HOME : VTAB 3: HTAB 20: REM POSITION FOR INPUT
20 IW = 2: GOSUB 1100: REM 2-CHAR INPUT FIELD
30 PRINT "-";
40 GOSUB 1100: REM 2-CHAR INPUT FIELD
50 PRINT "-";
60 GOSUB 1100: REM 2-CHAR INPUT FIELD
70 VTAB 3: HTAB 20: REM REPOSITION TO START OF INPUT FIELD
80 END
1090 REM ++ DISPLAY "IW" REVERSE-BLANKS ++
1100 INVERSE
1110 FOR I = 1 TO IW: PRINT " ";: NEXT I
1120 NORMAL
1130 RETURN

```

The program above includes statements that position the date entry to begin at column 21 on row 3. It also clears the screen so that residual garbage on the screen does not surround the request for a date. After displaying the date entry field, the cursor moves back to the first character position of the first entry field, although this is not apparent because of the END statement.

Try using an INPUT statement on line 80 to receive the first part of the date: the month. This could be done as follows:

```

80 INPUT M$
90 END

```

Enter statements on lines 80 and 90, as illustrated above, and execute them. As you see, the INPUT statement will not do. Apart from the fact that a question mark displaces the first reverse field character, pressing RETURN at the end of the

entry erases the rest of the display line. This is an occasion to use the GET statement. Add the following program lines:

```

80 GOSUB 1200:MM$ = C$: REM GET 1ST MONTH DIGIT
90 GOSUB 1200:MM$ = MM$ + C$: REM GET 2ND MONTH DIGIT
1190 REM ++ ACCEPT ONE INPUT CHARACTER ++
1200 GET C$
1210 PRINT C$:: REM ECHO KEYSTROKE
1220 RETURN

```

These statements accept a two-digit input. The input is displayed in the first reverse field of the date. The two-digit input needs no RETURN or other terminating keystroke. The program automatically terminates the data entry after two characters have been entered.

Three two-digit entries are needed: for the month, the day, and the year. Rather than repeating statements on lines 80 through 90, we will put these statements into a subroutine and go to it three times, as follows:

```

80 GOSUB 1300:MM$ = CC$: REM GET MONTH
90 PRINT "-": REM SPACE OVER TO NEXT FIELD
180 GOSUB 1300:DD$ = CC$: REM GET DAY
190 PRINT "-": REM SPACE OVER TO NEXT FIELD
280 GOSUB 1300:YY$ = CC$: REM GET YEAR
290 END
1290 REM ++ GET A 2-CHARACTER INPUT ++
1300 GOSUB 1200:CC$ = CC$ + C$: REM GET 2ND CHARACTER
1320 RETURN

```

The variables MM\$, DD\$, and YY\$ hold the month, day, and year entries, respectively. Each entry is held as a two-character string.

There are two ways in which we can help the operator recover from errors while entering a date.

1. The program can automatically test for valid month, day, and year entries.
2. The operator can be given a means of restarting the data entry.

The program can check that the month lies between 1 and 12. The program will not bother with leap years, but otherwise it will check for the maximum number of days in the specified month. Any year from 00 through 99 will be allowed. Any invalid entry will cause the entire date entry sequence to restart.

If the operator presses the RETURN key, then the entire date entry sequence restarts. Our final date entry program now appears as follows:

```

10 HOME : VTAB 3: HTAB 20: REM POSITION FOR INPUT
15 RC$ = CHR$ (13): REM ASSIGN RESTART-ENTRY CHARACTER
17 REM
18 REM DISPLAY THREE 2-CHAR. INPUT FIELDS
19 REM
20 IW = 2: GOSUB 1100
30 PRINT "-";

```

```

40 GOSUB 1100
50 PRINT "-";
60 GOSUB 1100
65 PRINT "": REM CTRL-G(BELL)
67 REM
68 REM GET THREE FIELD ENTRY
69 REM
70 VTAB 3: HTAB 20: REM REPOSITION TO START OF INPUT FIELD
79 REM
80 REM GET MONTH
81 REM
90 GOSUB 1300: IF C$ = RC$ THEN GOTO 10: REM CHECK FOR
ENTRY RESTART
100 MM = VAL (CC$): REM MONTH NUMBER
110 IF MM < 1 OR MM > 12 THEN 10: REM CHECK FOR UNREAL
MONTH
120 DT$ = CC$: PRINT "-": REM ADVANCE TO NEXT FIELD
125 REM DETERMINE MAX. NO. OF DAYS THIS MONTH
130 DM = 31: REM ASSUME 31 DAYS
135 REM UNLESS FEBRUARY
140 IF MM = 2 THEN DM = 29
150 REM OR APR, JUNE, SEPT, NOV
160 IF MM = 4 OR MM = 6 OR MM = 9 OR MM = 11 THEN DM = 30
169 REM
170 REM GET DAY
171 REM
180 GOSUB 1300: IF C$ = RC$ THEN GOTO 10: REM CHECK FOR
ENTRY RESTART
190 DD = VAL (CC$): IF DD < 1 OR DD > DM THEN 10: REM
RESTART IF ENTRY IS INVALID
200 DT$ = DT$ + "-" + CC$: PRINT "-": REM ADVANCE TO
NEXT FIELD
269 REM
270 REM GET YEAR
271 REM
280 GOSUB 1300: IF C$ = RC$ THEN GOTO 10: REM CHECK FOR
ENTRY RESTART
290 YY = VAL (CC$): IF YY < 0 OR YY > 99 THEN 10: REM
RESTART IF ENTRY IS INVALID
300 DT$ = DT$ + "-" + CC$
389 REM
390 REM DISPLAY ENTRY
391 REM
400 VTAB (10): HTAB (18): PRINT "DATE ENTERED:"
410 VTAB (11): HTAB (20): PRINT DT$
420 END
1089 REM
1090 REM ++ DISPLAY 'IW' REVERSE-BLANKS ++
1091 REM
1100 INVERSE
1110 FOR I = 1 TO IW: PRINT " ": NEXT I
1120 NORMAL
1130 RETURN
1189 REM
1190 REM ++ ACCEPT ONE CHARACTER INPUT ++

```



```

1191 REM
1200 GET C$: IF C$ = "" THEN 1200
1210 IF C$ = RC$ THEN RETURN : REM CHECK FOR RESTART
1220 IF C$ < "0" OR C$ > "9" THEN GOTO 1200
1230 PRINT C$;: REM ECHO KEYSTROKE
1240 RETURN
1289 REM
1290 REM ++ GET A 2-CHAR. INPUT ++
1291 REM
1300 REM GET 1ST CHARACTER; CHECK FOR RESTART
1310 GOSUB 1200: IF C$ = RC$ THEN RETURN
1315 CC$ = C$
1320 REM GET 2ND CHARACTER; CHECK FOR RESTART
1330 GOSUB 1200: IF C$ = RC$ THEN RETURN
1335 CC$ = CC$ + C$
1340 RETURN

```

Notice that the date is built up in the eight-character string DT\$, as month, day, and year are entered.

These three checks are made on data as it is entered:

1. Is the character a carriage return?
2. If the character is not a carriage return, is it a valid digit?
3. Is the two-character combination a valid month for the first entry, a valid day for the second entry, or a valid year for the third entry?

We selected the carriage return as a restart character. By replacing CHR\$(13) on line 15 you can select any other restart character. When the operator presses the selected restart key the entire date entry sequence restarts. We must check for the restart character in the one-character input subroutine (at line 1200) and again in the two-character input subroutine (at line 1300) since we want to be able to restart after the first or second digit has been entered. The main program also checks for a restart character in order to branch back to line 10 and restart the entire date entry sequence. You could branch directly out of the one-character input subroutine to line 10, thereby eliminating the other tests. But it is bad practice to exit a subroutine with a GOTO statement instead of a RETURN statement. Every subroutine should be treated as a logical module, with specified entry point(s) and standard subroutine returns. Branching out of a subroutine any other way inevitably leads to tangled programs and ultimately to program errors. (Remember that if you do branch out of the subroutine without using a RETURN statement, you must use the POP statement to clear the return location.)

Program logic that tests for nondigit characters resides entirely in the one-character input subroutine. We chose to ignore nondigit characters. The statement on line 1220 tests for nondigit characters.

Logic to check for valid month, day, and year must exist within the calling program (not the subroutine) since each of these two-character values has different allowed limits.

The statement on line 110 tests for a valid month.

Statements on lines 130, 140, and 160 compute the maximum allowed day

for the entered month. The statement on line 190 checks for a valid day.

The check for a valid year is very simple; it is on line 290.

It takes extra time to write a good data entry program that displays information in a pleasing manner and checks for valid data input, allowing the operator to restart at any time. Is the time worth spending? By all means yes. You will write a program once; an operator may have to run the program hundreds or thousands of times. Therefore you spend extra programming time once, in order to save operators hundreds or thousands of delays.

## FORMS DATA ENTRY

The following section describes some programming techniques that are best implemented in Applesoft. Many of the special effects we use in the example program in this section are difficult or impossible to obtain in Integer BASIC. If you are programming exclusively in Integer BASIC, you may still wish to read this section since some of the techniques presented in it are adaptable to Integer BASIC.

The best way of handling multiple-item data entry is to display a form, and then fill in the form as data is entered. Consider a name and address. First display a form as follows:

```

ENTER NAME AND ADDRESS BELOW
11 NAME:
22 STREET:
33 CITY:
44 STATE:
55 ZIP:
  
```

Notice that each entry has been assigned a number. The field numbers appear in reverse video on this form.

The operator enters data sequentially, starting with item 1 and ending with item 5. The operator can then change any specific data entry.

The following program will clear the screen and display the initial form:

```

109 REM
110 REM DISPLAY THE DATA ENTRY FORM
111 REM
120 CALL - 936: VTAB 2: REM CLEAR SCREEN AND POSITION
    CURSOR
125 PRINT "ENTER NAME AND ADDRESS BELOW"
130 REM FIRST DISPLAY FIELD NUMBERS
140 INVERSE
150 FOR I = 1 TO 4: HTAB 2: PRINT I: NEXT I
160 VTAB 6: HTAB 29: PRINT 5
170 NORMAL
180 REM NOW DISPLAY FIELD NAMES
190 VTAB 3: HTAB 6: PRINT "NAME:"
200 HTAB 4: PRINT "STREET:"
  
```

```

210 HTAB 6: PRINT "CITY:"
220 HTAB 5: PRINT "STATE:";
230 HTAB 31: PRINT "ZIP:"

```

As each data item is entered, we will create a reverse field to identify where data will appear as it is entered. CTRL-X is used to restart data entry in the current field. The RETURN key ends data entry in the current field. The following instruction sequence provides us with necessary program logic:

```

100 RC$ = CHR$ (24): REM CTRL-X IS THE RESTART CHAR.
299 REM
300 REM ENTER ALL 5 FIELDS
301 REM
310 FOR F = 1 TO 5: GOSUB 1900: NEXT F
320 END
990 REM ++++++SUBROUTINE 1000+++++
991 REM ENTER STRING DATA INTO A FIELD WITH LN CHARACTERS
992 REM THE CURSOR MUST BE IN THE FIELD'S FIRST POSITION
993 REM THE RETURN KEY WILL END DATA ENTRY
994 REM THE 'LEFT ARROW' KEY RESTARTS ENTRY
995 REM NO VALIDITY CHECKS ON ENTERED DATA
996 REM THE ENTERED STRING IS RETURNED IN CC$
997 REM
1000 HT = POS (0) + 1: REM REMEMBER START-OF-FIELD
    POSITION
1010 REM DISPLAY INVERSE VIDEO ENTRY MASK
1020 INVERSE
1030 FOR I = 1 TO LN: PRINT " ";: NEXT I
1040 NORMAL: HTAB (HT): REM REPOSITION TO START OF FIELD
1050 REM ENTER DATA
1060 CC$ = "": REM INITIALIZE OUTPUT TO NULL
1070 GET C$
1080 IF C$ = RC$ THEN HTAB (HT): GOTO 1020: REM RESTART
    ENTRY?
1090 IF C$ = CHR$ (13) THEN GOTO 1140: REM END OF ENTRY?
1100 REM WHEN ENTRY IS FULL, WAIT FOR RETURN OR RESTART
1110 IF LEN (CC$) = LN THEN GOTO 1070
1120 PRINT C$;: REM ECHO KEYSTROKE
1130 CC$ = CC$ + C$: GOTO 1070
1135 REM ENTRY FINISHED, FILL THE REST OF CC$ WITH BLANKS
1140 J = LEN (CC$)
1150 FOR I = J TO LN: CC$ = CC$ + " ": NEXT I
1160 REM REDISPLAY ENTRY
1170 HTAB (HT): PRINT CC$;: RETURN
1889 REM ++++++SUBROUTINE 1900+++++
1890 REM BRANCH TO ENTRY ROUTINE FOR FIELD NUMBER F
1891 REM
1900 ON F GOTO 2000,2100,2200,2300,2400: RETURN
1989 REM
1990 REM ENTER 20-CHAR. NAME
1991 REM
2000 VTAB 3: HTAB 11
2010 LN = 20: GOSUB 1000: NA$ = CC$: RETURN
2089 REM

```

```

2090 REM ENTER 20-CHAR STREET
2091 REM
2100 VTAB 4: HTAB 11
2110 LN = 20: GOSUB 1000: CI$ = CC$: RETURN
2189 REM
2190 REM ENTER 20-CHAR. CITY
2191 REM
2200 VTAB 5: HTAB 11
2210 LN = 20: GOSUB 1000: RETURN
2289 REM
2290 REM ENTER 18-CHAR. STATE
2291 REM
2300 VTAB 6: HTAB 11
2310 LN = 18: GOSUB 1000: ST$ = CC$: RETURN
2389 REM
2390 REM ENTER 5-CHAR. ZIP CODE
2391 REM

```

Key in the entire program from statement 100 to statement 2400 and run it. Remember if you still have statements 109 through 230 keyed into your computer from the last example, you do not need to reenter them.

If your program does not run correctly, check over your listing carefully. In particular, look for semicolons in PRINT statements.

When you run the program, each of the five fields will be highlighted in turn. As you enter characters they will echo in the field. When you hit the RETURN key the entire reverse field is replaced by the data you entered. Try using CTRL-X to restart data entry.

Carefully go through the logic of the string entry subroutine, beginning at line 1000 and ending at line 1170. Before going any further you should clearly understand this program logic.

Note how easy it is to see what you are entering, and how simple it is to restart any entry to correct errors.

After the complete name and address has been entered, the program should ask the operator if he or she wishes to make any changes; then the program should ask which field needs to be changed. A subroutine to ask a yes-or-no question appeared earlier in this chapter. We will use a modified version of it, where the calling program provides the question to be asked of the operator. Here is the complete program with added statements:

```

9 REM *****
10 REM THIS PROGRAM DISPLAYS A FORM FOR ENTERING A NAME AND
11 REM ADDRESS THEN IT REQUESTS ENTRY OF THAT DATA
12 REM *****
13 REM
100 RC$ = CHR$ (24): REM CTRL-X IS THE RESTART CHAR.
109 REM
110 REM DISPLAY THE DATA ENTRY FORM
111 REM
120 CALL - 936: VTAB 2: REM CLEAR SCREEN AND POSITION
CURSOR

```

```

125 PRINT "ENTER NAME AND ADDRESS BELOW"
130 REM FIRST DISPLAY FIELD NUMBERS
140 INVERSE
150 FOR I = 1 TO 4: HTAB 2: PRINT I: NEXT I
160 VTAB 6: HTAB 29: PRINT 5
170 NORMAL
180 REM NOW DISPLAY FIELD NAMES
190 VTAB 3: HTAB 6: PRINT "NAME:"
200 HTAB 4: PRINT "STREET:"
210 HTAB 6: PRINT "CITY:"
220 HTAB 5: PRINT "STATE:"
230 HTAB 31: PRINT "ZIP:"
299 REM
300 REM ENTER ALL 5 FIELDS
301 REM
310 FOR F = 1 TO 5: GOSUB 1900: NEXT F
319 REM
320 REM ALLOW CHANGES
321 REM
330 VTAB 23: HTAB 1: REM GET ENTRY ON BOTTOM LINE
340 QU$ = "DO YOU WANT TO MAKE ANY CHANGES? "
350 GOSUB 1300: REM GET Y/N RESPONSE
360 IF YN$ = "N" THEN GOTO 500
370 VTAB 23: HTAB 1: REM GET ENTRY ON BOTTOM LINE
380 QU$ = "ENTER NUMBER OF FIELD TO CHANGE "
390 LO = 1: HI = 5
400 GOSUB 1400: REM GET NUMERIC RESPONSE
410 F = NM: GOSUB 1900: REM CHANGE FIELD F
420 GOTO 330
490 REM END OF PROGRAM
500 VTAB 23: HTAB 1: GOSUB 1200: REM CLEAR BOTTOM LINE
510 END
990 REM ++++++SUBROUTINE 1000+++++
991 REM ENTER STRING DATA INTO A FIELD WITH LN CHARACTERS
992 REM THE CURSOR MUST BE IN THE FIELD'S FIRST POSITION
993 REM THE RETURN KEY WILL END DATA ENTRY
994 REM THE 'LEFT ARROW' KEY RESTARTS ENTRY
995 REM NO VALIDITY CHECKS ON ENTERED DATA
996 REM THE ENTERED STRING IS RETURNED IN CC$
997 REM
1000 HT = POS (0) + 1: REM REMEMBER START-OF-FIELD POSITION
1010 REM DISPLAY INVERSE VIDEO ENTRY MASK
1020 INVERSE
1030 FOR I = 1 TO LN: PRINT " ";: NEXT I
1040 NORMAL: HTAB (HT): REM REPOSITION TO START OF FIELD
1050 REM ENTER DATA
1060 CC$ = "": REM INITIALIZE OUTPUT TO NULL
1070 GET C$
1080 IF C$ = RC$ THEN HTAB (HT): GOTO 1020: REM RESTART
    ENTRY?
1090 IF C$ = CHR$ (13) THEN GOTO 1140: REM END OF ENTRY?
1100 REM WHEN ENTRY IS FULL, WAIT FOR RETURN OR RESTART
1110 IF LEN (CC$) = LN THEN GOTO 1070
1120 PRINT C$;: REM ECHO KEYSTROKE
1130 CC$ = CC$ + C$: GOTO 1070
1135 REM ENTRY FINISHED, FILL THE REST OF CC$ WITH BLANKS

```

```

1140 J = LEN (CC$)
1150 FOR I = J TO LN:CC$ = CC$ + " ": NEXT I
1160 REM REDISPLAY ENTRY
1170 HTAB (HT): PRINT CC$:: RETURN
1189 REM ++++++SUBROUTINE 1200+++++
1190 REM CLEAR ROW WHICH THE CURSOR IS ON
1191 REM
1200 HTAB 1: REM START AT BEGINNING OF ROW
1210 FOR I = 1 TO 39: PRINT " ": NEXT I
1220 HTAB 1: REM LEAVE CURSOR AT BEGINNING OF ROW
1230 RETURN
1289 REM ++++++SUBROUTINE 1300+++++
1290 REM ASK A QUESTION (QU$) AND RETURN A Y OR N RESPONSE
    IN YN$
1291 REM
1300 GOSUB 1200: REM CLEAR ENTRY LINE
1310 PRINT QU$:: REM DISPLAY PROMPT
1320 GET YN$: IF YN$ < > "N" AND YN$ < > "Y" THEN GOTO
    1320
1330 PRINT YN$:: REM ECHO RESPONSE
1340 RETURN
1389 REM ++++++SUBROUTINE 1400+++++
1390 REM ASK FOR NUMERIC ENTRY (PROMPT IS QU$)
1391 REM RETURN RESPONSE IN NM
1392 REM NM MUST BE <=HI AND >=LO
1393 REM
1400 GOSUB 1200: REM CLEAR ENTRY LINE
1410 PRINT QU$:: REM DISPLAY PROMPT
1420 GET C$:NM = VAL (C$)
1425 REM CHECK THAT ENTRY IS WITHIN RANGE
1430 IF NM < LO OR NM > HI THEN GOTO 1420
1440 PRINT C$:: REM ECHO RESPONSE
1450 RETURN
1889 REM ++++++SUBROUTINE 1900+++++
1890 REM BRANCH TO ENTRY ROUTINE FOR FIELD NUMBER F
1891 REM
1900 ON F GOTO 2000,2100,2200,2300,2400: RETURN
1989 REM
1990 REM ENTER 20-CHAR. NAME
1991 REM
2000 VTAB 3: HTAB 11
2010 LN = 20: GOSUB 1000:NA$ = CC$: RETURN
2089 REM
2090 REM ENTER 20-CHAR STREET
2091 REM
2100 VTAB 4: HTAB 11
2110 LN = 20: GOSUB 1000:CI$ = CC$: RETURN
2189 REM
2190 REM ENTER 20-CHAR. CITY
2191 REM
2200 VTAB 5: HTAB 11
2210 LN = 20: GOSUB 1000: RETURN
2289 REM
2290 REM ENTER 17-CHAR. STATE
2291 REM
2300 VTAB 6: HTAB 11

```

```

2310 LN = 17: GOSUB 1000:ST$ = CC$: RETURN
2389 REM
2390 REM ENTER 5-CHAR. ZIP CODE
2391 REM
2400 VTAB 6: HTAB 35
2410 LN = 5: GOSUB 1000:ZI$ = CC$: RETURN

```

You should study the name and address program carefully-and understand the data entry aids which have been included. They are:

1. By labeling each field and juxtaposing a reverse video entry mask at the appropriate time, you clearly indicate to the operator what data is expected, and how many entry spaces are available.
2. When an operator enters the number of a field to change, the reverse field mask again quickly tells the operator whether the correct field number was specified.
3. An operator does not have to fill in all the characters of a field; when the operator presses the RETURN key the balance of the field is filled out with blank characters.
4. At any time the operator can restart entry in a field with CTRL-X.
5. When the program asks questions, only meaningful character responses are recognized: Y or N for "yes" and "no," or a number between 1 and 5 to select a field. It is bad practice to allow any entry other than a meaningful one. For example, to recognize Y for "yes" and *any* other character for "no" could be disastrous, since accidentally tapping a key could take the operator out of the current data entry prematurely. Conversely, recognizing N for "no" and any other character for "yes" would cause the operator to unnecessarily reenter data into some field, just because the operator accidentally touched the wrong key.

These are data entry features which we have not included but could add:

1. Check the ZIP code for any nondigit entry. (Note that similar codes in some countries do allow alphanumeric entries.)
2. Many cautious programmers will ask the question ARE YOU SURE? when an operator answers NO in response to the question DO YOU WANT TO MAKE ANY CHANGES? This gives the operator a second chance in the event that she or he accidentally touched the wrong key.
3. Provide an additional key which aborts a current data entry and restores the prior value. For example, if the operator chooses the wrong field to change, the example program forces the operator to reenter the field. The program could easily recognize a key which aborts the current data entry and retains the previous entry.
4. Enable the ← key for use as a backspace key. Each time the ← key is pressed, the cursor backs up one space and the last-entered character is replaced by an entry mask character (reverse video) on the display

screen and by a blank character in the subroutine output string (CC\$). Of course there can be no backspace when the cursor is at the left edge of the entry field.

Try modifying the name and address entry program yourself to include the additional safety features described above.

## FORMATTING OUTPUT

When you turn on an Apple II computer, output automatically goes to the display screen. There are statements which will send the output to a printer or any other device capable of receiving output.

There are a number of differences in programming output to a screen display as compared to a printer. For example, the printer may be wider than the display, in which case output which will fit on a printed line would run over the display line. On the other hand, HTAB and VTAB (TAB in Integer BASIC) can be used to move the cursor around the screen display, but they cannot be used to move a print head around on a piece of paper.

There are also many similarities in the programming techniques used to create printer and display screen output. Most of the discussion that follows applies to both. Any that applies only to display screens is noted. If you are planning to write programs that output to a printer you should also read the discussion of printer programming given later in this chapter.

Programming output is much simpler than programming data entry, since there is no operator interaction to worry about. You must make sure that the information is easy to use, and that is all. Here are a few rules to follow:

1. Avoid crowding too much information into a very small space.
2. If numbers or character strings are listed in columns, align the data so that the eye can quickly run down the column.
3. Use reverse fields to highlight key information, top headings, and side headings (display screen only).

Below are some hints that will help you avoid unnecessary errors when programming output.

1. Remember to follow individual items in a PRINT statement with a semicolon unless you specifically want the spacing provided by commas. This is the most common source of errors in output programming.
2. Before doing anything else, design your display screen or report. Use a piece of graph paper or a form made specifically for report or display design. Appendix L has a display screen that will allow you to compute rows and columns accurately. The alternative to advance planning is trial and error, which in the end will take a lot more time than drawing the display or report first.



3. Watch for array subscripts which do not divide evenly into columns. For example, suppose you have 25 items in array N\$(I) which you are printing in three columns. You might be tempted to do something like this:

```

100 FOR I = 1 TO 25 STEP 3
200 REM PROCESS COLUMN 1
.
.
.
300 REM PROCESS COLUMN 2
.
.
.
400 REM PROCESS COLUMN 3
.
.
.
500 NEXT I

```

But on the final pass of the FOR-NEXT loop, indexes 26 and 27 will be computed, although they do not exist. You can easily check for the end of an array in a FOR-NEXT loop as follows:

```

100 FOR I = LO TO HI STEP ST
.
.
.
350 I = I + 1
360 IF I > HI THEN 510
.
.
.
500 NEXT
510 REM CONTINUE WITH PROGRAM

```

### The Display Screen as a Data Window

When dealing with large quantities of data, a very common technique is to use the display screen as a window on the data. At any time the display screen shows only part of the data available. One way of doing this is to group the data into pages, each of which will fit on the display screen. Programs that use this technique must have separate routines to display the field headings and data values for each page. They must also provide some means by which the program user can switch from one page to another.

Very often arrays are used to hold large amounts of data. In this case, you can



The actual column numbers will appear where you see XX above. The actual row numbers will appear where you see YY. Here are the necessary program statements to create row and column headings in reverse video:

```

1000 INVERSE
1020 FOR I = 1 TO 3
1030 HTAB 4 + I * 10: PRINT "COLUMN";
1040 NEXT I
1050 PRINT
1060 FOR I = 0 TO 2
1065 REM 1 EXTRA SPACE AHEAD OF 1-DIGIT NOS.
1070 SZ = 0: IF CZ + I < 10 THEN SZ = 1
1080 HTAB 18 + I * 10: PRINT SPC(SZ);CZ + I;
1090 NEXT I
1100 PRINT
1110 FOR I = RZ TO RZ + 9
1115 REM 1 EXTRA SPACE AHEAD OF 1-DIGIT NOS.
1120 SZ = 0: IF I < 10 THEN SZ = 1
1130 HTAB 4: PRINT "ROW"; HTAB 8: PRINT SPC(SZ);I
1140 NEXT I
1150 NORMAL : RETURN

```

We deliberately created a window that is smaller than the entire screen so that we can better illustrate the concept of a window on data. There is nothing to stop you creating a window that occupies your entire screen; however there will be occasions when you want a small window so that other data can appear on the screen concurrently.

We will now add instructions that ask the operator to enter two numbers representing the smallest column and row of the array. The array element with this column and row number will appear in the top lefthand display position. The display window will be filled with array elements from adjacent columns and rows, up to the end of the window. Here is the complete program:

```

5 REM WINDOW AN A TABLE DISPLAY PROGRAM
6 REM *****
10 HOME : PRINT "PLEASE WAIT...INITIALIZATION IN PROCESS";
20 DIM XZ(14,50)
30 FOR I = 1 TO 14
40 FOR J = 1 TO 50
50 XZ(I,J) = I * 100 + J
60 NEXT J
70 NEXT I
75 HOME
80 HTAB 1: VTAB 20: INPUT "ENTER COLUMN (1 TO 12):";CZ
90 IF CZ < 1 OR CZ > 12 THEN GOTO 80
100 VTAB 21: INPUT "ENTER ROW (1 TO 41):";RZ
110 IF RZ < 1 OR RZ > 41 THEN GOTO 100
120 VTAB 1: HTAB 1: GOSUB 1000: REM DISPLAY HEADINGS
130 REM FILL IN WINDOW VALUES
135 VTAB 3
140 FOR I = RZ TO RZ + 9
150 HTAB 10
160 FOR J = CZ TO CZ + 2
165 REM DISPLAY RIGHT-JUSTIFIED VALUES IN WINDOW

```

```

170 X$ = STR$(X%(J,I)): PRINT SPC(10 - LEN(X$));X$;
180 NEXT J
190 PRINT : REM NEXT DISPLAY LINE
200 NEXT I
210 VTAB 22: PRINT "CONTINUE? ENTER Y OR N ";
220 GET C$: IF C$ < > "Y" AND C$ < > "N" THEN 220
230 IF C$ = "Y" THEN GOTO 80
240 END
990 REM
991 REM ++++++SUBROUTINE 1000+++++
992 REM DISPLAY ROW AND COLUMN HEADINGS
1000 INVERSE
1020 FOR I = 1 TO 3
1030 HTAB 4 + I * 10: PRINT "COLUMN";
1040 NEXT I
1050 PRINT
1060 FOR I = 0 TO 2
1065 REM 1 EXTRA SPACE AHEAD OF 1-DIGIT NOS.
1070 S% = 0: IF C% + I < 10 THEN S% = 1
1080 HTAB 18 + I * 10: PRINT SPC(S%);C% + I;
1090 NEXT I
1100 PRINT
1110 FOR I = R% TO R% + 9
1115 REM 1 EXTRA SPACE AHEAD OF 1-DIGIT NOS.
1120 S% = 0: IF I < 10 THEN S% = 1
1130 HTAB 4: PRINT "ROW";: HTAB 8: PRINT SPC(S%);I
1140 NEXT I
1150 NORMAL : RETURN

```

Enter this program into the computer and run it. If you enter the program correctly, the first thing you will notice is that the computer stops and appears to do nothing for a while; it is executing the nested FOR-NEXT statements occurring on lines 30 through 70. It takes five or ten seconds to fill array X% with numbers. The program displays an advisory message about the initialization. Without such a message, the program user may well assume that the computer is not working. It is a good idea to display a prominent message whenever such periods of apparent inactivity occur.

Note that column numbers from 1 through 12 are allowed. There are three columns, therefore any column number up to 12 will stay within the array dimension of 14 columns. Row numbers from 1 to 41 are allowed, since ten column numbers starting with 41 would run through 50, which is the other array dimension.

The integer value from array X% is converted into a string on line 170 before being printed. We made this conversion to simplify display formatting. It is then easy to compute the number of spaces between columns, as shown by the PRINT statement on line 170. It is not so easy to align numeric values correctly when displaying them directly. To see this for yourself, change line 170 as follows:

```
170 PRINT SPC <7>; X% <J, I>;
```

Numbers will align providing you do not display any four-digit numbers — at which time the display will be too wide for a 40-character screen.

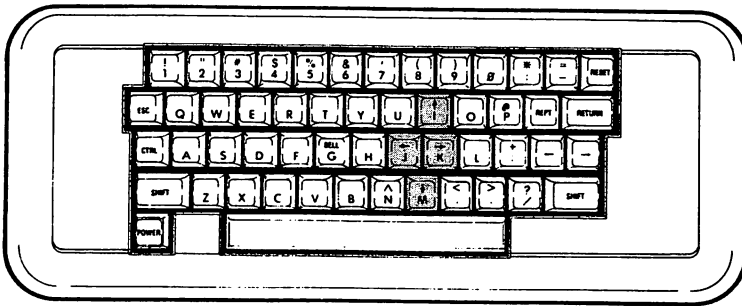
Our program takes great care to terminate the display on the 39th column of

the display, rather than the 40th and last column. If you run displays out to the 40th column, you will run afoul of the wrap-around logic whereby lines that are more than 40 characters long automatically continue on the next line. You are best off not tangling with the display formatting nightmare that can result from the interaction between carriage returns generated by printing in column 40 and your own formatting carriage returns.

As an exercise, it is worth modifying the complete table display program so that it does go out to the 40th column. To do this you must change the horizontal tab on line 150 from 10 to 11, on line 1030 from  $4+I*10$  to  $5+I*10$ , on line 1080 from  $18+I*10$  to  $19+I*10$ , and finally on line 1130 from 4 and 8 to 5 and 9. Now try running the program; the columns of numbers line up, but you have too many carriage returns and the column numbers in the top headings are covered by the first row of array values. See if you can eliminate the extra carriage returns and generate the correct display. This is not an easy programming task.

Notice that the statements which ask for input on lines 80, 100, and 220 are all followed by program steps that don't allow invalid inputs. Even in this simple demonstration program we take the time to program safe input.

A useful refinement to a program that displays a window on an array is to provide the operator with means of moving the window up or down one row, or right or left one column. This is easily done. We will use the I, J, K, and M keys for directional control in much the same manner as they are used in edit mode (described in Chapter 3). The I key moves up one row, M down one row, J left one column, and K right one column as shown below.



To accomplish this task we must replace lines 210 through 240 with the following statements:

```

210 VTAB 22: PRINT "CONTINUE?"
215 PRINT "ENTER DIRECTION (I,J,K,M), Y, OR N ";
220 GET C$
225 REM DOWN ONE ROW?
230 IF R% > 1 THEN IF C$ = "M" THEN R% = R% - 1: GOTO 120
235 REM UP ONE ROW?
240 IF R% < 41 THEN IF C$ = "I" THEN R% = R% + 1: GOTO 120
245 REM LEFT ONE COLUMN?
250 IF C% > 1 THEN IF C$ = "J" THEN C% = C% - 1: GOTO 120
255 REM RIGHT ONE COLUMN?

```

```
260 IF C% < 12 THEN IF C$ = "K" THEN C% = C% + 1: GOTO 120
270 IF C$ = "Y" THEN GOTO 80: REM ENTER NEW ROW AND COLUMN
280 IF C$ = "N" THEN END
285 REM SOUND BELL AND REJECT ANY OTHER ENTRY
290 PRINT CHR$(7): GOTO 220
```

Notice how straightforward the logic is, even though we are still checking for operator errors. Any entry other than one of the six allowed characters is rejected, and a directional control character is rejected if it would move the window past the edge of the array dimensions.

## PROGRAMMING PRINTERS

The Apple II treats the printer as a substitute for the screen display. In order to create printer output, therefore, you must include program statements that deflect output from the display to the printer. Output must be deflected back to the display when you have finished printing. This is done using the PR# statement.

Printers connect to computers via a serial or parallel interface, depending on the printer.

Normally, serial interface cards are inserted in slot 1 of an Apple II computer, while parallel interface cards are inserted in slot 2. But this is a convention rather than a necessity. In fact, serial and parallel interface cards can be inserted in any slot (other than slot 0).

### Outputting Text to a Printer

You may recall that PR# is considered a DOS statement whenever DOS is present. This means it must be printed with a prefix character of CTRL-D (ASCII code 4). The program below will print two lines of text on a printer connected to an interface card in slot 1 of an Apple II computer with DOS present:

```
10 REM OUTPUT TWO LINES OF TEXT TO A PRINTER
20 REM CREATE A CTRL-D CHARACTER
30 D$=""
40 REM SELECT THE SERIAL I/O PORT
50 PRINT D$;"PR#1"
60 PRINT "TO A SCREEN OR THE PRINTER AN APPLE WILL WRITE"
70 PRINT "AND IN EACH CASE THE DATA GOES OUT BYTE BY BYTE"
80 REM DESELECT THE PRINTER
90 PRINT D$;"PR#0"
100 END
```

The statement on line 30 creates the control character which converts the PR# command into a BASIC statement. PR# statements appear on lines 50 and 90.

The statement on line 50 deflects output from the screen to the printer, whereas the statement on line 90 deflects output back to the screen. The PRINT statements on lines 60 and 70 output two lines of text to the printer. Here is the output created when the program is run:

```
TO THE SCREEN OR A PRINTER THE APPLE WILL WRITE
AND IN EACH CASE THE DATA GOES OUT BYTE BY BYTE
```

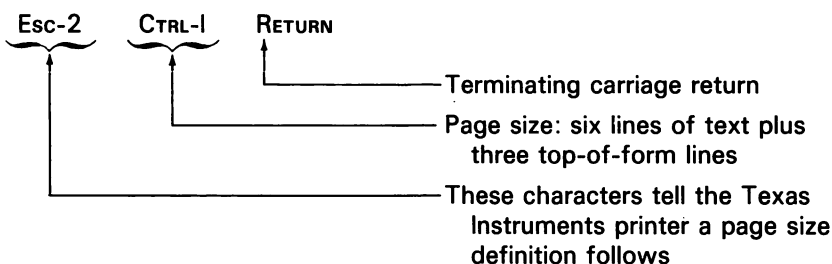
Here is the same program for an Apple II with DOS absent:

```
10 REM OUTPUT TWO LINES OF TEXT TO A PRINTER
40 REM SELECT THE SERIAL I/O PORT
50 PRINT "PR#1"
60 PRINT "TO A SCREEN OR THE PRINTER AN APPLE WILL WRITE"
70 PRINT "AND IN EACH CASE THE DATA GOES OUT BYTE BY BYTE"
80 REM DESELECT THE PRINTER
90 PRINT "PR#0"
100 END
```

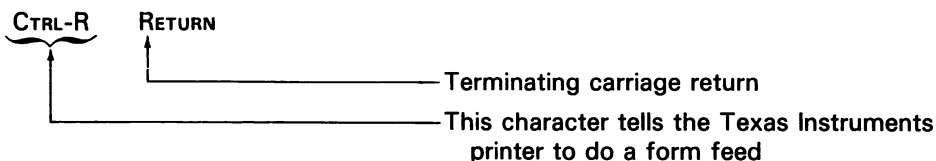
### Programmable Printers

Many printers allow output to be formatted under program control. By including appropriate control characters in the text data stream going out to the printer, you can adjust line lengths, character set, page length, and a variety of other printed page characteristics.

A variety of popular printers are commonly used with Apple II computers. From a programming viewpoint, however, the Apple II treats all printers identically. For example, suppose a Texas Instruments model 810 printer is connected to an Apple computer via a serial interface in slot 1. The Texas Instruments manual says we can set the page size to six lines per page with this character sequence:



In addition, we can force the printer to advance to the top of the next page (this is called a *form feed*) with this character sequence:



Modifying our earlier printer program, the Applesoft listing below outputs the same two lines of text 15 times, in six-line blocks. The program shown is for use with DOS present.

```

10 REM   OUTPUT TEXT TO THE PRINTER
11 REM   USING 6 LINES PER PAGE AND 5 PAGES
20 REM   CREATE A CTRL-D CHARACTER
30 D$ = ""
40 REM   SELECT THE SERIAL I/O PORT
50 PRINT D$;"PR#1"
51 REM   SELECT 6 LINES PER PAGE
52 PRINT CHR$(27);"2"; CHR$(6)
53 PRINT CHR$(12);: REM PRINT FORM FEED TO POSITION FOR
  START OF OUTPUT
54 FOR I = 1 TO 15
60 PRINT "TO A SCREEN OR THE PRINTER AN APPLE WILL WRITE"
70 PRINT "AND IN EACH CASE THE DATA GOES OUT BYTE BY BYTE"
75 NEXT I
80 REM   DESELECT THE PRINTER
90 PRINT D$;"PR#0"
100 END

```

In the program above, the statement on line 52 selects six lines per page. The CHR\$(27) function represents the Esc character. CHR\$(9) defines a page as being six lines plus three lines which separate the bottom of one page from the top of the next. The semicolons concatenate the three characters to give the required character stream.

The statement on line 53 executes the top of form. The CHR\$(12) function is the form feed character. A semicolon follows the form feed character since without it the PRINT statement would produce a carriage return, and that would result in a blank line. Without the semicolon, therefore, the first page would have one blank line and five printed lines on it, while subsequent pages would have six printed lines per page.

Any other programmable printer options are enacted by outputting other appropriate nonprinting *control codes* in the same way.

This program will not work in Integer BASIC because it uses the CHR\$ function. You can generate almost any string value by pressing some key or combination of keys, though. So you can replace CHR\$ with a pair of quotation marks enclosing a non-printing keystroke. For example, the string value generated by entering (as three keystrokes) "Esc" is the same as CHR\$(27). See Appendix I for a complete list of such equivalents.

### Printing Program Listings

If you type the LIST command at the keyboard, any program in the Apple II memory will be listed at the display. To *print* the listing instead, enter appropriate PR# commands before and after issuing the LIST command. Assuming that the



printer is connected to an interface card in slot 1, here are the required steps:

1. Make sure that the program to be listed is in the Apple II memory.
2. With the cursor on a blank screen line, select the printer by typing PR#1. The cursor will return to the first character of the current line but will not drop down to the next display line.
3. Now type LIST. This command will not appear on the screen, but it will show up on the printer along with a printout of the program to be listed.
4. When the entire program has been listed, return output to the display screen by typing PR#0. This command will not appear on the display screen; it may appear on the printer.

## STORING DATA ON CASSETTE

We learned in the last chapter how to save and load programs on the cassette tape. In Applesoft, you can also record numeric and integer arrays on cassette tape.

The STORE instruction records array values and the RECALL instruction reads them back in. Neither statement controls tape movement, nor do they display directives telling the program user when to operate the cassette recorder buttons. The program below demonstrates the use of STORE and RECALL. It assigns values to a numeric array, stores the array on a cassette tape, then sets all the array values to zero, and finally recalls the array values from the tape. The array values are displayed at key points to document the changes that occur as the program progresses.

```

10 REM THIS PROGRAM DEMONSTRATES STORE AND RECALL
20 REM *****
30 DIM A(10)
40 HOME
50 PRINT TAB( 4);"STORED"; TAB( 13);"CLEARED"; TAB( 22);
   "RECALLED"
60 REM INITIALIZE ARRAY VALUES
70 FOR I = 1 TO 10:A(I) = I: NEXT I
80 T = 8: GOSUB 1000: REM DISPLAY VALUES TO BE STORED
90 VTAB 20: HTAB 1
100 PRINT "PLACE CASSETTE IN RECORDER. REWIND IT."
110 PRINT "PRESS THE 'RECORD' AND 'PLAY' BUTTONS."
120 INPUT "AND ENTER 'GO' ";C$
130 IF C$ < > "GO" THEN GOTO 90
140 STORE A
160 CLEAR : REM SET ARRAY VALUES TO ZERO
170 VTAB 2:T = 18: GOSUB 1000: REM DISPLAY CLEARED ARRAY
180 VTAB 20: HTAB 1
190 GOSUB 1100: REM ERASE LAST INSTRUCTIONS
200 VTAB 20: HTAB 1
210 PRINT "REWIND TAPE. PRESS 'PLAY' BUTTON,"

```

```

220 INPUT "AND ENTER 'GO' ";C$
230 IF C$ < > "GO" THEN GOTO 200
240 RECALL A
260 VTAB 2:T = 28: GOSUB 1000: REM DISPLAY RECALLED VALUES
265 VTAB 20: HTAB 1
270 GOSUB 1100: REM ERASE LAST INSTRUCTIONS
280 VTAB 20: HTAB 1
290 PRINT "PRESS 'STOP' BUTTON."
300 END
990 REM ++++++SUBROUTINE 1000+++++
991 REM DISPLAY VALUES OF ARRAY A
1000 FOR I = 1 TO 10: HTAB T: PRINT A(I): NEXT I: RETURN
1090 REM ++++++SUBROUTINE 1100+++++
1091 REM ERASE THREE DISPLAY LINES
1100 FOR I = 1 TO 119: PRINT " "; NEXT I: RETURN

```

You can store an array under one variable name and recall it using a different variable name. But generally speaking, the dimensions of the array you store with must be the same as the dimensions of the array you recall with. There are some complicated exceptions which are covered in Chapter 8. Unless you are specifically trying to achieve some special effects, use RECALL with data from any given cassette using an array with the same dimensions as the one you used STORE with.

## PROGRAM OPTIMIZATION

Traditionally, the optimal program is the one that, for a given task, runs the fastest and uses the least memory. Of course this dual goal must be moderated so that the resulting program is still reliable, easy to write, easy to use, easy to read, and easy to change. You will benefit more in the long run by spending your time directly on these aspects of your programs instead of tweaking programs for maximum speed and minimum memory requirements. Still, if you know how to optimize program speed and memory usage, you can initially write programs that are efficient and don't need any fine tuning after they're running. In this spirit, we present a few ways to write programs that are faster and use less memory.

Some of the techniques for making a program run faster will make it take more space, while some ways of decreasing space requirements will increase program execution time. You will have to decide which is more important in your particular program.

### FASTER PROGRAMS

Avoid using constants (e.g., 0, 100, "Y", "ENTER"). Instead, assign the value of the constant to a variable early in your program. Then use the variable where you would have used the constant. This is especially important when you are repeatedly using constant integer values in real expressions. It takes longer to convert a constant to a real value than it does to look up the value of a variable. And

when such a conversion takes place inside a FOR-NEXT loop, an often-used subroutine, or a user-defined function, the difference becomes that much more significant. This technique has the added benefit of making your program easier to change. If you should ever need to change the constant, it will be easier to change the one assignment statement than to hunt down and change every occurrence of the constant.

Use those variables that are referenced often in a program as early in the program *execution* as possible. Memory space for variables is allocated on a first come, first serve basis. BASIC will find a variable at the front of the list faster than one at the end of the list.

When BASIC encounters an instruction to branch to another line number, it starts looking for that line number at the beginning of the program and searches sequentially through the program until it finds it. Clearly, the lower a line number is in relation to the rest of the lines in the program, the faster BASIC can branch to it. Therefore, assign the lowest line numbers in your program to the most often-used subroutines.

In Applesoft, do not include index variables with NEXT statements. That way, Applesoft does not have to verify whether you specified the correct index.

## COMPACT PROGRAMS

Use subroutines to avoid duplicate programming for identical logic. This will also go a long way towards improving the readability, reliability, and changeability of your program.

Use the zero elements of arrays (e.g., X(0), B(0)).

There are fewer characters in a short variable name than there are in a constant value that has many digits. So assign constant values to variable names and use the variable names in place of the constant values.

Put more than one statement on a program line. It takes an extra five bytes of memory for each extra program line. Note however that compound program lines are hard to edit and harder yet to read and understand. Figuring out how to make the program work the first time is bad enough; it's even worse to have to do it time and time again.

Use REM statements judiciously; abbreviate comments. But be careful; the fewer remarks your program has, the harder it will be to understand when you come back to it later on.

Be thrifty with the use of variables. Each variable requires a certain amount of memory, even if you only use it once. So establish a system of assigning variable names that includes some *scratch* variables which can be used for FOR-NEXT loops, intermediate calculations, and the like. Don't overdo it though; your program will be easier to understand if you assign variables in a meaningful way. Establish standard identities for individual variables (e.g., CN\$ is the customer name) and groups of variables (e.g., all scratch variables start with X).

Use INPUT statements and data files (if you have a disk drive; see Chapter 5)

instead of assignment statements and DATA statements.

In Applesoft, use integer arrays instead of real arrays. Each integer array value takes two bytes of memory, while real array values require five bytes each. Use the FRE function periodically in your program to clean up the string storage area of memory.

## DEBUGGING

A new program never seems to work quite the way you expect it to. Even if there are no errors in the BASIC syntax, there are likely to be errors in the program logic. Either kind of error is a *bug*. The process of finding and eliminating program errors is called *debugging*. There are several approaches you can take to debugging a program.

This is an appropriate place for the usual caveat: take your time, plan it out, get it right the first time. Don't sit down at the keyboard with a half-baked notion about what you want your program to do and start hacking away. If you are new to programming, supplement this book with one of the BASIC primers listed in Appendix K to get some pointers on good programming practices.

If you have written your program and it doesn't seem to be working right and you can't figure out why, there are some BASIC statements you can use that will help you debug your program.

### The PRINT Statement

Surprisingly, the plain old PRINT statement is a very useful debugging tool. You can temporarily put extra PRINT statements in your program at strategic points to display messages to tell you that the program has reached a certain point without failing, and to print out intermediate values of variables. This way you can trace the flow of program execution and you can check the results of intermediate calculations.

### The TRACE Statement

The TRACE statement lives up to its name; it traces the flow of program execution by displaying the line number of each statement as it is executed. To see how it works, type in the following program, then type TRACE followed by RUN.

```
100 PRINT "ENTER A NUMBER FROM 1 TO 5 (6 TO END)";  
110 INPUT N  
120 IF N = 1 THEN PRINT "UNO";  
130 IF N = 2 THEN PRINT "DOS";  
140 IF N = 3 THEN PRINT "TRES";  
150 IF N = 4 THEN PRINT "CUATRO";
```

```

160 IF N = 5 THEN PRINT "CINCO";
170 IF N > 5 THEN GOTO 100
180 FOR I = 1 TO N
190 PRINT " *": REM PRINT N ASTERISKS
200 NEXT I
210 CALL - 936: REM CLEAR SCREEN
220 GOTO 100

```

To cancel TRACE mode and return to normal, execute the statement NO TRACE.

### The DSP Statement

There is another useful debugging statement available only in Integer BASIC: the DSP statement. Here is an example:

```
>10 DSP COUNT
```

Once this particular DSP statement has been executed, the Apple II will notify you each time the value of variable COUNT changes, and on which line number the change occurred. Since the RUN command disables any previously executed DSP statements, you must use GOTO to start the program, or put your DSP statements on program lines.

You can also turn off DSP mode for a variable with a NO DSP statement. Here is an example:

```
>300 NODSP NAME$
```

Once this statement has been executed, the Apple II ceases to notify you each time the value variable NAME\$ changes.

Try adding the following lines to the example we used with TRACE to see the effect of DSP. Although you can use both TRACE and DSP simultaneously, try running the example program without TRACE to more clearly see the effects of DSP.

```

10 DSPN
20 DSP I
100 PRINT "ENTER A NUMBER FROM 1 TO 5 (6 TO END)";
110 INPUT N
120 IF N = 1 THEN PRINT "UNO";
130 IF N = 2 THEN PRINT "DOS";
140 IF N = 3 THEN PRINT "TRES";
150 IF N = 4 THEN PRINT "CUATRO";
160 IF N = 5 THEN PRINT "CINCO";
170 IF N > 5 THEN GOTO 100
180 FOR I = 1 TO N
190 PRINT " *": REM PRINT N ASTERISKS
200 NEXT I
210 CALL - 936: REM CLEAR SCREEN
215 NODSPN
220 GOTO 100

```

## IMMEDIATE AND PROGRAMMED MODE RESTRICTIONS

You can use most BASIC statements in deferred mode or as commands in immediate mode. There are some statements that are legal only in deferred mode, and others that are legal only in immediate mode. Table 4-2 lists the restricted statements for Integer BASIC. Table 4-3 lists the restricted statements for Applesoft.

TABLE 4-2. Integer BASIC Statements Restricted to  
Immediate or Deferred Mode

Deferred Mode Only	Immediate Mode Only
END FOR GOSUB INPUT NEXT RETURN	AUTO CLR CON DEL HIMEM: LOAD LOMEM: MAN NEW RUN SAVE

TABLE 4-3. Applesoft BASIC Statements Restricted to  
Immediate Mode

Immediate Mode Only
DATA DEF FN GET INPUT ON ERR GOTO RESUME

# 5

## The Disk II

The disk drive is one of the most important components of a computer system. Disk drives allow almost instantaneous access to any item in a large block of information. The Apple Disk II can store more than 100,000 characters of data on a single diskette. That is over twice the amount that can be stored in 48K of RAM, and when the computer is turned off, all of the information stored on a diskette remains intact.

### ABOUT DISKS

Disks store information magnetically, the same way a tape recorder does. The biggest difference is that a disk is round, like a record. It spins like a record too. Inside the disk drive there is a *head* which can read and write information. The computer can move the head to any location on the surface of the disk. This ability is called *random access*. Thus, the disk is a *random access storage device*. A special program, called the *Disk Operating System*, or *DOS*, controls all the operations of the disk. There are several different kinds of disks.

### Hard Disks

Hard disks are rigid, and coated with a magnetic substance. Hard disks typically store 5 or 10 megabytes of data (a megabyte equals one million bytes). Most hard

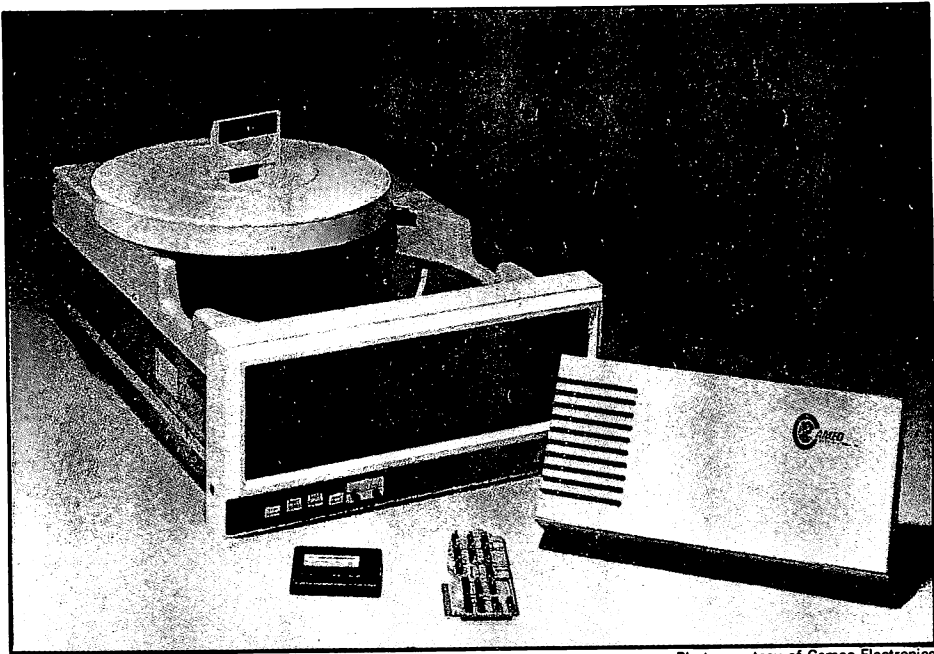
**FIGURE 5-1. Typical Hard Disk System**

Photo courtesy of Cameo Electronics

disks are removable; that is, the disk and the drive are separate, so you can change disks. Hard disks cost about \$150.00 each; hard disk drives cost \$3,000 to \$10,000. Figure 5-1 shows a typical hard disk system.

### **Winchester Disks**

Winchester disk drives, pictured in Figure 5-2, use a special technology that allows six to ten times more data to be stored on each disk. Winchester disks are extremely susceptible to dust and dirt — even cigarette smoke. Because they must be kept very clean, Winchester disks are sealed inside the disk drive and cannot be changed. Winchester disk systems cost from \$2,500 to \$8,000.

### **Diskettes**

Diskettes are the most popular type of disk. A diskette consists of a circular vinyl disk enclosed in a rigid plastic envelope. The envelope protects the diskette from damage during normal handling and use. The diskette spins freely inside the envelope. Openings in the envelope allow the head access to the surface of the diskette and provide an area where the drive can grip and spin the diskette. The diskette should never be removed from its envelope. Figure 5-3 shows what a diskette looks like outside its envelope.



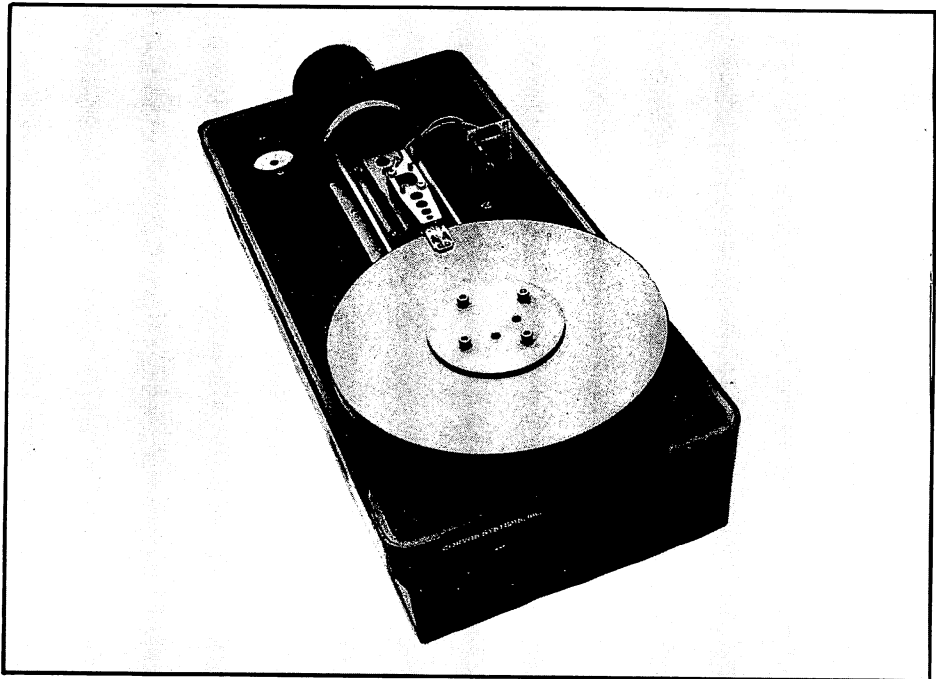


FIGURE 5-2. Winchester Disk Drive Photo courtesy of Corvus Systems, Inc.

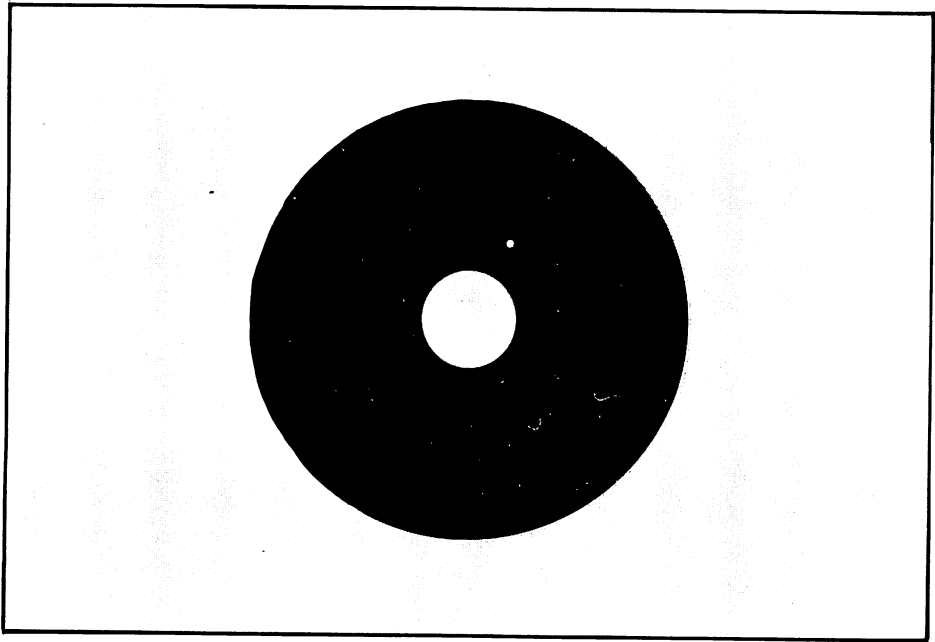


FIGURE 5-3. Diskette Without Protective Envelope

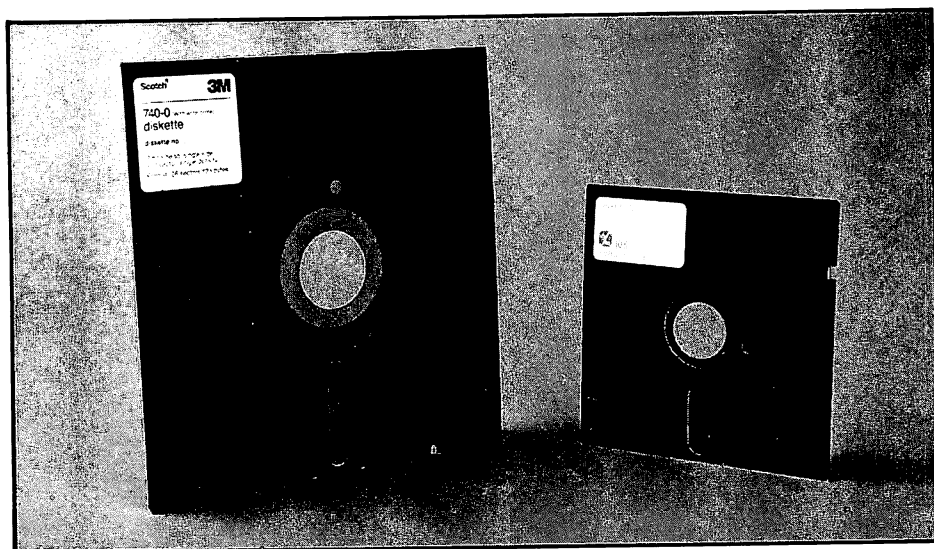


FIGURE 5-4. 8-inch and 5-1/4-inch Diskettes

Diskettes, also known as floppy disks, come in two sizes: 8-inch diameter and 5-1/4-inch diameter, both illustrated in Figure 5-4. Apple Disk II uses 5-1/4-inch diskettes, which are also called *mini-disks* or mini-diskettes. The Disk II can store over 100,000 bytes on each diskette.

## HOW DATA IS STORED ON DISKS

You should be familiar with the various facets of the disk storage process. Information stored on a disk is the result of many coordinated actions.

### Tracks

To find one particular byte quickly among the more than 100,000 bytes stored on each diskette, Apple DOS divides a diskette into 35 tracks, numbered 0 to 34. Tracks are similar to grooves on a record, except that you cannot see them, and they do not connect; that is, tracks are concentric circles, one inside another, as shown in Figure 5-5.

### Sectors

To further speed up the search for a particular byte, DOS divides each track into 16 *sectors*, as Figure 5-6 shows. Each sector holds exactly 256 bytes. Armed with the track and sector of a particular byte, DOS has only 256 bytes to search. Now accessing a particular byte, or series of bytes, is almost instantaneous. The real power of the disk drive is utilized.

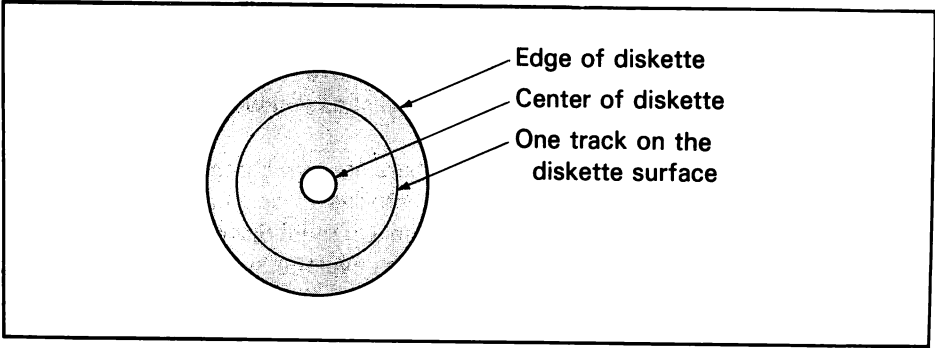


FIGURE 5-5. Diskette Tracks

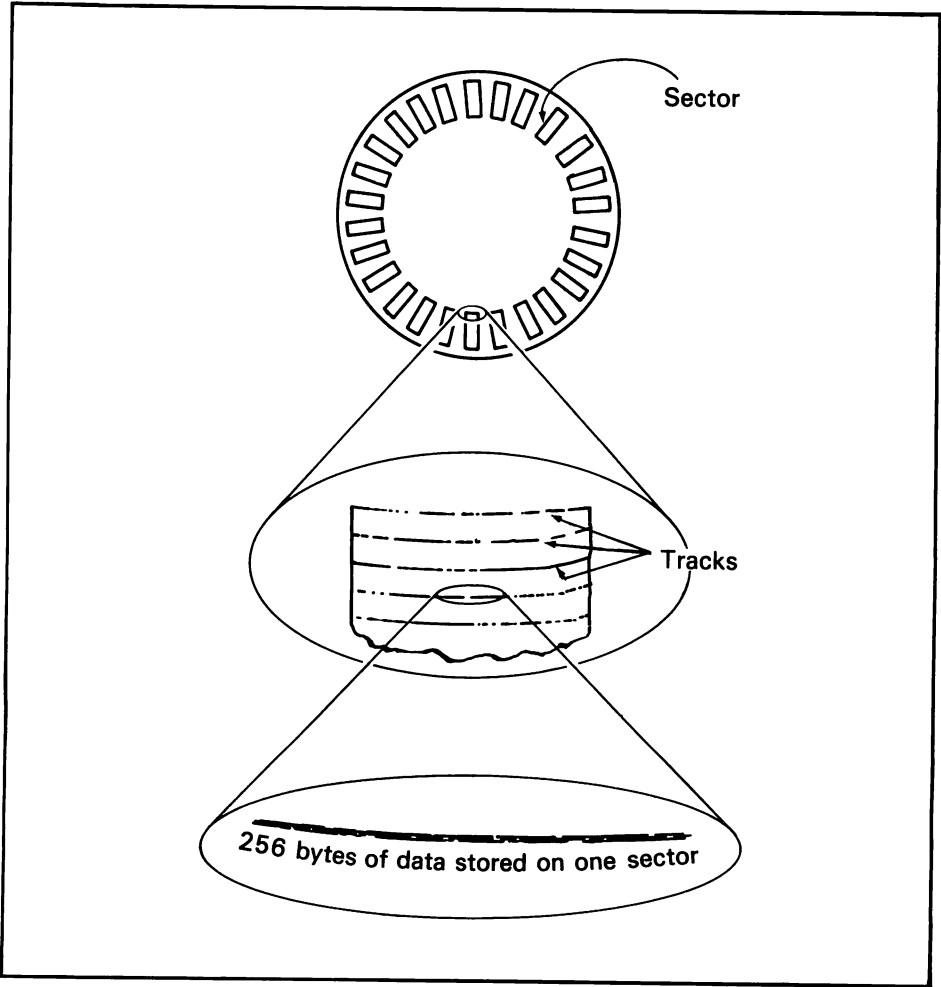


FIGURE 5-6. A Diskette's Recorded Surface

## LOCATING TRACKS AND SECTORS

Finding a track on a diskette is simple: the drive moves the head to the point on the diskette where the desired track is located, just as you would go about selecting a particular song on a record album.

Finding a sector is a little more difficult. There are two common methods used to locate specific sectors on diskettes. Both methods utilize a hole, called an *index hole*, which is punched in the diskette envelope. On most diskettes the index hole is located near the large hole in the center of the envelope. As the diskette spins, a hole (or holes) located on the diskette itself passes the hole in the envelope. A light source inside the drive passes through to a sensor whenever the holes are aligned. The computer senses pulses of light and computes sector locations based on this information.

There are two methods used to locate diskette sectors, called *hard sectoring* and *soft sectoring*.

### Hard Sectors

Hard sectored diskettes have several holes, as shown in Figure 5-7. Each hole indicates the location of a sector. An extra hole marks the location of the first sector. The computer locates sectors by counting holes following the first sector.

### Soft Sectors

Soft sectored diskettes have just one index hole, as Figure 5-8 shows. It marks the first sector. Locations of other sectors are computed by timing the floppy disk's rotation. The Apple Disk II uses soft sectored diskettes.

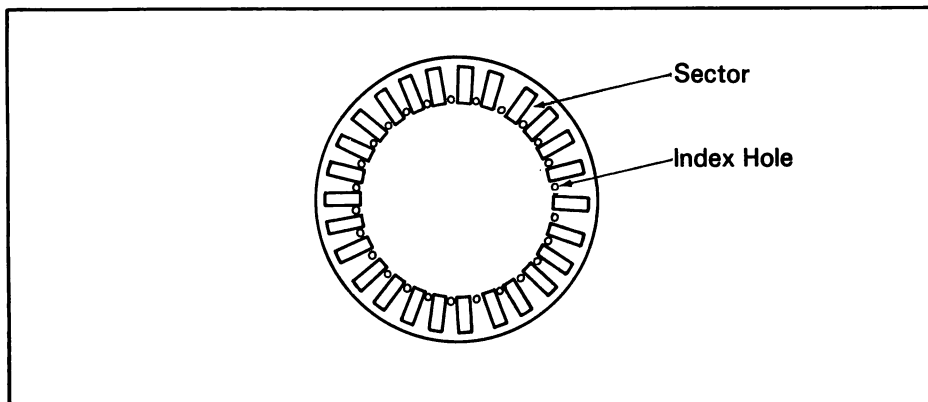


FIGURE 5-7. Hard Sectored Diskette

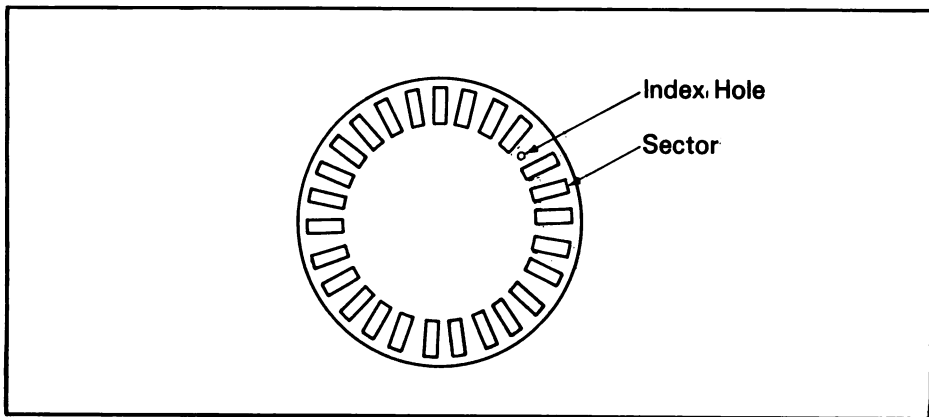


FIGURE 5-8. Soft Sectored Diskette

### WRITE PROTECTING

There is also a notch on the side of the envelope. This notch is used to enable or prevent information being written on the diskette. On 8-inch diskettes this notch is a *write protect* notch, because the computer will not write on a diskette if it *has* a notch on the diskette cover. On 5-1/4-inch diskettes the notch is called the *write enable* notch, because the computer will not write on a diskette *unless* the notch exists. Some diskettes, such as the "System Master Diskette" supplied with the Disk II, are protected permanently. They do not have a notch. Notched 5-1/4-inch diskettes may be protected by covering the notch with a piece of tape. Figure 5-9 shows how this works.

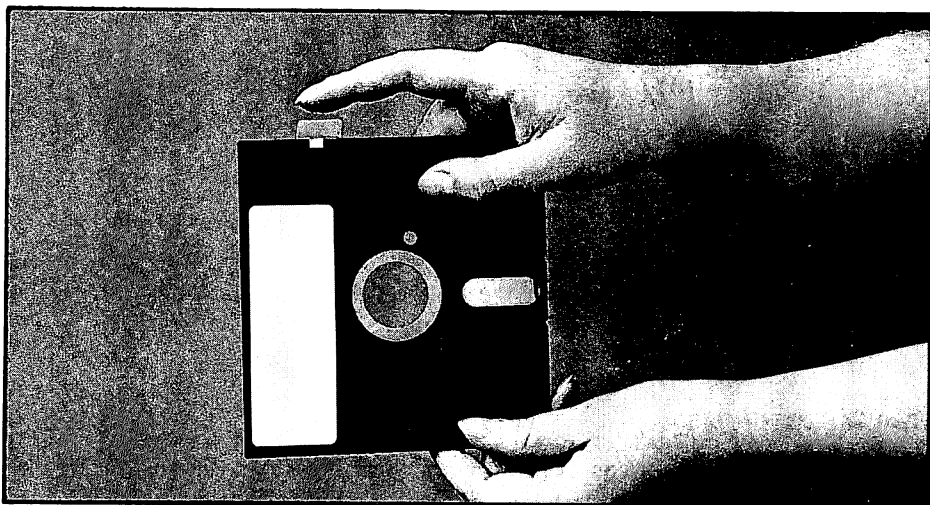


FIGURE 5-9. Write Protecting a 5-1/4-inch Diskette

## THE DISK OPERATING SYSTEM

All disk-related operations are controlled by a special program called the *Disk Operating System*, or DOS. BASIC transmits requests to DOS for any operation involving the disk. The DOS returns the results to BASIC.

### VERSIONS OF DOS

Several versions of DOS now exist. DOS 3.3 is the most recent; this chapter describes it. The chief difference between DOS 3.3 and DOS 3.2.1, the next most recent version, is the number of sectors they establish on a disk. DOS 3.2.1 has 13 sectors, while DOS 3.3 has 16. Apple Computer Inc. has a special program which converts disks from DOS 3.2.1 to DOS 3.3.

### INITIALIZING DISKS

Before Apple II can use a diskette, the diskette must be *initialized*. Initializing a diskette erases all files from the diskette and places a copy of DOS in tracks 0, 1, and 2. Disk II initialization instructions are given later.

### DISK FILES

Information is stored on diskettes as *files*. A file can have any length that can be physically accommodated by the diskette. Every file has a name. A file may hold information like text, a computer program, or an image of an Apple II graphics display. The various types of files are discussed in more detail later.

### DISKETTE DIRECTORY

The name of every file on a diskette is stored in the diskette's *directory*. The directory is located in track 17 of the diskette. The first entry in the directory is in sector 15. The last entry is in sector 1. The directory has enough room to index up to 84 files.

Stored with each file's name is a code indicating the type of data in the file, the number of sectors it occupies, and the location of the sector that contains the file's *track/sector list*.

### TRACK/SECTOR LIST

The track/sector list contains pairs of bytes which specify the track and sector address of each sector used by the file. Each pair of bytes is called a *link*. The first link in a track/sector list gives the address of the next sector used by the list itself.

The list may occupy as many sectors as it needs. The second link is the address of the first sector used by the file itself, the third link addresses the next sector used by the file, and so on. A link of zero marks the end of the list.

### Overview of the Disk Storage Process

DOS controls the flow of all data to and from a diskette. When you write to a file, several things happen:

1. DOS searches the diskette directory for the name of the file. (Your program names the file in order to identify it, as we will describe later.)
2. If the name is found, DOS reads 256 bytes from the proper sector and stores them in an area of memory called a *buffer*. If the name is not found, or if that sector of the file has never been written to before, DOS fills the buffer with zeros.
3. Up to 256 bytes of the data being written to the file are copied from memory into the buffer. If less than 256 bytes of data are copied into the buffer, prior data remains.
4. If, and only if, 256 bytes are written to the buffer, buffer contents are written back to the proper diskette sector.
5. This process (steps 3 and 4) repeats until all of the data being written to the file has been copied into the buffer and stored on the diskette.
6. After all the data has been written to the diskette, DOS updates the track/sector list and the directory.

Note that unless the number of bytes you write to the file is evenly divisible by 256, the last block of bytes will not fill the buffer, and steps 4 through 6 will never happen. Therefore a Disk II command called the CLOSE command forces data in the buffer to be written to the file, after which the track/sector list and directory are updated. This process is called *closing* the file. Failure to close a file after writing to it can result in loss of data. Always close a file when you have finished with it. The CLOSE command is discussed in more detail later.

### DISK CRASH

One of the worst disk errors that may occur is called a disk *crash*. There are two types of disk crashes: *hard* crashes and *soft* crashes.

A hard crash occurs when the surface of the diskette is damaged, or has a physical defect, like a rip or a piece of dirt. A hard crash can cause damage to the read/write head inside the disk drive. The damaged head can, in turn, damage more disks. For this reason, always handle disks with care.

A soft crash results when the directory track is overwritten with incorrect data. This most frequently occurs when one or more files have been written to but not closed, a different disk is placed in the drive, then the files from the original diskette are closed. To fully appreciate the resulting mess you must experience it.

## BOOTING THE DISK II

For the Apple II to recognize any disk command, the special Disk Operating System program (DOS) must be in memory. If you had a lot of time on your hands, you could type DOS into memory via the keyboard. But there is an easier way: it is called *booting* the disk. Booting the disk, or booting DOS, reads a copy of DOS from a disk and places it in memory.

### HOW TO BOOT DOS

There are several different ways to boot DOS, depending on the configuration of your computer and the language you use to initiate the boot. Each method assumes that the disk drive is connected to the DRIVE 1 pins on the disk controller card in slot 6.

Insert the "System Master Diskette" into the drive and close the drive door. What you do next depends on the Apple system you are using. At the end of a successful boot, the display screen will look like one of those in Figure 2-5.

#### Autostarting

Autostarting is the easiest way to boot DOS. As the name implies, booting is automatic. But to autostart, your computer must have the Autostart Monitor. You can tell if you have the Autostart Monitor installed by turning on the computer while the Disk II is connected. If the red IN USE lamp on the Disk II cabinet lights and the drive makes whirring and clicking noises, your Apple II has the Autostart Monitor.

To boot with an Autostart Monitor, simply turn on the Apple II power switch.

#### Booting from the Monitor

When the Monitor prompt character (\*) appears on the screen, the Apple II Assembly Language Monitor is waiting to accept commands. There are several different ways of booting DOS from the Monitor.

#### Monitor Jump Booting

Type the letter C, followed by the slot number of the drive you wish to boot (6 is standard), then two zeros and the letter G. The entire jump boot command should look like this:

\*C600G

C600 is the memory address of the program which boots from the drive in slot #6. G is the command that transfers control to that program. Now press RETURN.



The lamp on the drive should light and the drive will make whirring and clicking noises.

### **CTRL-K and CTRL-P Monitor Boots**

The other Monitor commands you can use to boot the DOS are CTRL-K and CTRL-P. To boot from the Monitor using either of these two commands, type the slot number (normally 6), then type CTRL-K or CTRL-P. The CTRL-K or CTRL-P will not be displayed on the screen.

After typing the command, press the RETURN key.

### **Booting from Integer BASIC or Applesoft**

The same boot commands are recognized by both Integer BASIC and Applesoft.

### **Booting From BASIC Using the PR# and IN# Commands**

After the BASIC prompt character (> in Integer BASIC, or ] in Applesoft), type the letters PR or IN, then a pound sign (#), and finally the slot number of the disk you want to boot. The command should look like one of these:

PR#6  
or  
IN#6

Now press the RETURN key.

### **Booting with the Apple Language System**

If you have the Language System installed in your Apple II, the booting procedures above may not apply. It takes two disks to boot DOS versions 3.2.1, 3.2, and lower. But DOS 3.3 does boot as described above.

To boot DOS 3.2.1, 3.2, and lower using the Language System, insert the diskette labeled "BASICS: Integer and Applesoft II" (supplied with the Language System) in place of the "System Master Diskette" described above, then proceed with any boot procedure described above (autostart, PR#6, etc.). After a successful boot, the screen will display:

INSERT BASIC DISK AND PRESS RETURN

This is quite misleading. Actually you must insert a DOS diskette (the "System Master Diskette" will do, or you can use any diskette that has been initialized). After you have inserted the second diskette and pressed RETURN, booting will proceed normally and you will see one of the displays shown in Figure 2-5.

## BEGINNING DISK COMMANDS

The Apple II Disk Operating System interprets and executes disk commands. Many commands allow or require additional parameters that further define the operation to be executed. A few elementary commands are described below.

### CATALOG

The CATALOG command outputs all the file names in the diskette directory to the current output device, usually the screen.

The catalog first shows the diskette volume number. (Volume numbers are discussed later.)

For each file on the disk, CATALOG lists the *type* of data in the file, whether or not the file is *locked*, the *number of sectors* the file occupies, and the *file name*. A typical catalog is shown in Figure 5-10.

### File Types

The type of data in a file is represented by the single-letter code which appears in the leftmost column of the catalog. The codes used are listed in Table 5-1.

### Locked Files

Locked files may not be written to or deleted. The catalog indicates that a file is locked by preceding the type code with an asterisk (\*). If a space appears instead of an asterisk, the file is not locked. Locked files are discussed later in this chapter.

```
*I 002 HELLO
*I 053 APPLE-TREK
*I 018 ANIMALS
*B 009 UPDATE 3.2.1
*I 014 COPY
*I 009 COLOR DEMO
*I 053 BRICK OUT
*I 026 SPACE WAR
*I 050 THE INFINITE NO. OF MONKEYS
*I 051 COLOR SKETCH
*I 053 SUPERMATH
*I 026 APPLEVISION
*I 017 BIORHYTHM
*I 027 PINBALL
```

FIGURE 5-10. Typical Disk Catalog

TABLE 5-1. Disk File Type Codes

Code	Meaning
A	Applesoft programs
B	Binary image files
I	Integer BASIC programs
T	Text files

### Number of Sectors

The number of sectors the file occupies is shown as a three-digit number. The smallest (empty) files are one sector long. If a file has more than 255 sectors, this number is reset to 0 and starts over. This does not affect the true size of the file.

### File Names

Apple II DOS requires that you refer to a file by its name. Here are the rules which govern the names you assign to files:

1. File names must be from 1 to 30 characters in length. Excess characters are ignored.
2. File names must begin with a letter.
3. Any character you can type on the keyboard may be part of a file name, except commas.

You may use nonprinting characters (like those created with the `CTRL` key) as part of a file name, but they will not show up in the catalog listing. This is useful if you want to prevent others from knowing your file names. (But do not forget which nonprinting characters you used.)

### Using the CATALOG Command

To use the `CATALOG` command, simply type it in (assuming DOS has been booted) like this:

`CATALOG`

The result should look something like Figure 5-10.

If the number of lines in the `CATALOG` printout exceeds 20, the computer will display the first 21 lines and wait until you press any key on the keyboard (except `RESET`, `CTRL`, and `SHIFT`); then it displays the next 21. This pause gives you time to read all the file names before they scroll off the top of the screen.

## LOAD

The LOAD command reads a program file from the diskette into memory. You must specify the name of the file to be loaded. This example loads the program named COLOR DEMOSOFT:

```
LOAD COLOR DEMOSOFT
```

If the file name you specify is not in the diskette directory, you will get the error message FILE NOT FOUND.

If the file is on the diskette, DOS checks the type of data in the file. If it is not a program file, you will get the error FILE TYPE MISMATCH.

Assuming everything is OK, LOAD erases the program currently in read/write memory, then copies the program from the file into read/write memory. After the prompt and cursor return you may list, modify, or run the newly loaded program.

## THE DISK VERSION OF THE RUN COMMAND

Frequently after you issue the LOAD command, you immediately issue the RUN command. You can abbreviate this two-step process by specifying a file name with the RUN command. The LOAD command becomes an implicit step, since the file must be loaded before it can be run. Here are a few examples:

```
RUN PROGRAM 2  
RUN SPOT RUN  
RUN COLOR DEMOSOFT
```

## SPECIFYING THE DRIVE NUMBER

Many DOS commands allow a disk drive to be specified. Two parameters specify a diskette drive: the *drive* parameter and the *slot* parameter.

Two disk drives can be connected to one diskette controller. To select a drive, add a comma and D1 or D2 to a diskette command as follows:

```
LOAD UP,D2  
RUN AROUND,D1  
CATALOG,D2
```

After you have used a drive specification once, subsequent disk commands will default to that same drive until you specify the other drive. The default drive is always the drive specified by the most recent command. If none has yet been specified, drive 1 is used.

## SLOT SPECIFICATION

The Disk II controller cards plug into slots inside the Apple II. There are eight slots available, but slot number 0 cannot be used for Disk II controllers. That leaves a

maximum of seven slots for Disk II controllers. Since each controller will support one or two disk drives, a maximum of 14 drives may be connected to an Apple II.

If you have more than two Disk II drives connected to your Apple II, you cannot refer to them as D3 or D4, etc. Instead, you must use another parameter to select the proper controller card. The parameter is called *slot* and is used to indicate which slot the disk controller you want to reference is plugged into.

To use the slot parameter, you add a comma, the letter S, and the slot number (1-7) to disk commands, like this:

```
CATALOG, S5  
LOAD TRUCKS, S6  
RUN OVER, S3
```

The slot used when booting DOS becomes the default slot. This slot will be selected until another slot is selected by a slot parameter.

After you have used a slot parameter once, subsequent disk commands will default to that same slot until you specify another slot.

You may use both the drive and the slot parameters together, in the same command. By specifying a slot number from 1 to 7, and a drive number of 1 or 2, you can refer to any Disk II connected to your computer. The slot and drive parameters can appear in any order. The following two commands are equivalent:

```
CATALOG, D2, S5  
CATALOG, S5, D2
```

Both commands will produce the catalog of the diskette in drive number 2 of the controller in slot number 5.

### Problems with the Slot Parameter

If you select a slot which does not have a controller in it, the computer will lock up. The computer is waiting for the nonexistent controller to signal it is ready. In order to recover with your program intact, press **RESET**. If you get a BASIC prompt (> or ]), everything should be OK. If you get the Monitor prompt (\*), type **3DOG** and press **RETURN**. If that doesn't work, you will lose your program because you will have to reboot DOS.

### VOLUME SPECIFICATION

*Volume* is another parameter you may specify with every DOS command that allows the slot and drive specifications, except **CATALOG**. Volume allows you to make sure the correct diskette is in the drive you have selected.

The **CATALOG** command ignores volume references. When the **CATALOG** command lists the diskette directory, the volume number of the disk is the first item listed.

To use the volume parameter, add a comma and the letter V followed by a volume number, as follows:

```
LOAD ZONE, V191
```

If the volume number you specify is not the same as the volume number assigned to the diskette when it was initialized, DOS returns the error message **VOLUME MISMATCH**.

Volume numbers must be in the range 1 to 254. If you do not specify a volume number, or if you specify a volume number of 0, the volume parameter is ignored.

You may use the volume parameter in conjunction with either the slot or drive parameter, or both, in any sequence. Here are some examples:

```
LOAD CARGO,D2,V24
RUN PAYROLL,S6,V111
```

## MORE DISK II COMMANDS

You now know how to see what is on a disk using **CATALOG**, and how to load and run programs. You are ready to put your own programs on disks, using the commands **INIT**, **SAVE**, **DELETE**, **LOCK**, **RENAME**, and **VERIFY**, which are described below.

### INIT

Before you can write on a diskette, remember that it must first be *initialized*. When a diskette is initialized, anything stored on it is erased, so make sure you do not initialize a diskette containing data you want to save.

The **INIT** command stores on the diskette any program that is in memory when you use the **INIT** command. This becomes the *greeting* program, which is automatically run every time you boot from that diskette. The greeting program can be as simple or as complex as you desire. For example, suppose you have a diskette that contains a mailing list. You could use the mailing list program as the greeting program. Then when you put the diskette in the drive and boot, presto! The mailing list program is up and running. Another example is a greeting program that consists of a **NEW** statement and an **END** statement. Every time the disk is booted, you will get the BASIC prompt (**>** or **]**).

A well written greeting program should tell you something about the diskette. A typical greeting program might look like this:

```
100 TEXT
200 CALL - 936
300 PRINT "THIS IS MY FIRST DISKETTE."
400 PRINT
500 PRINT "INITIALIZED 2/16/81"
600 PRINT
700 PRINT "ON A 48K SYSTEM USING DOS 3.3"
800 END
```

The file name of the greeting program must be specified when you use the **INIT** command. It is your responsibility to make sure there is always a program by that

name on the diskette. If you delete the greeting program (we'll describe how later), you will see the error message FILE NOT FOUND every time you boot from that diskette. The only way to stop that error message is to put a program on the diskette with the greeting program's file name. That might be difficult to do if you do not know, or cannot remember what file name was assigned, because there is no way of determining what the name of the greeting program is. The best solution is prevention. *Always* specify the same greeting program file name when you initialize diskettes. The standard greeting program name is HELLO.

### Using the INIT Command

A typical INIT command looks like this:

```
INIT HELLO,S6,D1,V36
```

As you might suspect, the slot, drive, and volume specifications are optional. If you include the volume parameter, DOS *assigns* the specified volume number to the diskette. INIT is the only command that assigns the volume number. When the volume parameter is used with other DOS commands, the number specified must match the number assigned by INIT. If you omit the volume parameter, INIT will assign a volume number of 254.

As always, omitting the slot or drive parameters will cause them to default to the values specified by the previous DOS command. Be sure you know which drive the INIT command is going to select. If you are not positive, specify!

To initialize a new diskette, first remove the "System Master Diskette" from the drive and replace it with a new, blank diskette. Use the NEW command to make way in memory for the greeting program. Then type in the greeting program shown above, or a greeting program of your own. It is a good idea to test run the greeting program before it is stored on the diskette.

Let's assign a volume number of 123. Now type:

```
INIT HELLO,S6,D1,V123
```

Make sure the drive door is shut, and press RETURN. The red lamp on the drive should light, accompanied by the usual whirring and clicking sounds. The entire process takes about two minutes, so be patient. After the lamp goes out, use the CATALOG command to see what is on the new disk. The result should look like this:

```
DISK VOLUME 123
```

```
I 002 HELLO
```

or, if you are using Applesoft, you should see:

```
DISK VOLUME 123
```

```
A 002 HELLO
```

The letter I means the greeting program was written in Integer BASIC. The letter A means it was written in Applesoft. The 002 means the program is two sectors long. You might see a different number, depending on the length of your

greeting program.

Prepare a label for your diskette. You should write the volume number on the label, along with any information you may want to know about this particular diskette. Remove the diskette from the drive, apply the label, and replace the diskette in the drive. If you want to reboot using the diskette you have just prepared, you can do so now.

## SAVE

If you have kept up with the hands-on part of this chapter, you will now have a freshly initialized diskette in drive 1, slot 6 of your Apple II. You probably also have a copy of your greeting program in memory. Use the LIST command and see if it is there. If it is not, type:

```
LOAD HELLO
```

to get a new copy from the disk.

The SAVE command stores programs on a diskette. To save another copy of your greeting program, type the SAVE command followed by a file name. For this example, we will use the file name GREETING PROGRAM. You may use any valid file name you wish.

```
SAVE GREETING PROGRAM
```

The disk should spin and make its usual racket. When SAVE is finished, the BASIC prompt and cursor will reappear. Next, use the CATALOG command to check the disk contents. What you see should look something like this:

```
DISK VOLUME 123  
  
A 002 HELLO  
A 002 GREETING PROGRAM
```

You can save any BASIC program you desire this way.

If you use a file name that is already on the diskette, whatever program you are saving will replace the program with the same name that was stored on the diskette. Thus the old program is automatically erased. This works as long as the old program and the new program are in the same version of BASIC. If they're not, the new program will not be saved. Instead, the message FILE TYPE MISMATCH appears.

## DELETE

After a while, you will probably accumulate many programs on a diskette which no longer serve any useful purpose. The DELETE command removes files from the diskette.

To delete the copy of the greeting program you just saved, any of the following commands will do the job (unless you have referenced a different disk drive since then).



```
DELETE GREETING PROGRAM,S6,D1,V12  
DELETE GREETING PROGRAM,V123  
DELETE GREETING PROGRAM
```

Remember that you can use the slot, drive, and volume parameters in any order you wish, or not at all if the default drive is the one you want to access.

## LOCK

Some program or data files on a diskette must be kept permanently. For this purpose, DOS supports a protective technique called file *locking*. Locking a file prevents it from being accidentally deleted or written over. To lock a file, enter the LOCK command, followed by the file name and the optional drive, slot, and volume parameters. The following command locks the greeting program:

```
LOCK HELLO
```

It is a good idea to LOCK the greeting program.

Subsequent attempts to delete or write to locked files will result in the error message FILE LOCKED.

If the locked file is an Integer BASIC or Applesoft program and you try to save a program that has the same name but is written in the other BASIC, then you will get the error FILE TYPE MISMATCH.

On a diskette catalog, locked files are indicated by an asterisk (\*) preceding the file type.

## UNLOCK

When you decide to write over or delete a locked file, you can remove the lock with the command UNLOCK. The following statement unlocks the greeting program:

```
UNLOCK HELLO
```

As always, slot, drive, and volume specifications are optional and can occur in any order.

## RENAME

You can change the name of any file on a diskette. One way to change the name of a program file would be to load it, delete it, then save it with the new name. A better way is to use the RENAME command. RENAME works for any file on the disk, regardless of which BASIC you are using. RENAME also works for text files and binary files. Here is an example of RENAME:

```
RENAME OLDNAME,NEWNAME
```

DOS will *not* check to see if the new file name is already on the diskette. If it is, you will end up with two files that have the same name. This can be very confusing and difficult to recover from.

Do not RENAME the greeting program unless you put a new program with the old name on the diskette. RENAME does not change the name DOS looks for when the diskette is booted.

You cannot change the name of a file which is locked.

You may specify slot, drive, and volume numbers (in any order) if you wish.

## VERIFY

Occasionally, you may want to check that a file is intact. The best way to do this is using the VERIFY command. The following command will check out the greeting program:

```
VERIFY HELLO,V123
```

As DOS writes out each sector of a file, it computes a number, referred to as a *checksum*. The checksum value is based on the numeric value of each character in the sector. The checksum is stored with the sector. When you issue the VERIFY command, DOS recalculates the checksum for each sector of the file and compares the computed checksums with the stored checksums. If there is any difference between the values, DOS returns the error message I/O ERROR.

If all the computed values match the stored values, DOS does not return any message; the BASIC prompt and cursor simply reappear.

As with most other DOS commands, you may specify slot, drive, and volume numbers in any order if you wish.

## USING DOS COMMANDS IN PROGRAMS

Until now all DOS commands have been typed in on the Apple II keyboard. But BASIC programs that use files can include DOS commands.

To use a DOS command from within a BASIC program, you put it in a PRINT statement, prefixed with an ASCII code 4 character. Use CTRL-D to create this prefix. CTRL-D must be the first character output by the PRINT statement. Be sure that the previous PRINT statement did not terminate with a semicolon or a comma.

Since CTRL-D produces a nonprinting character, it is recommended that you document each occurrence with a REM statement, like this:

```
1000 PRINT "RUN MENU" : REM THERE IS A CTRL-D  
      BETWEEN THE FIRST QUOTE AND THE R IN RUN
```

You can simplify things by defining a variable as CTRL-D, then printing the variable followed by the DOS command. The variable D\$ is commonly used for this, but you can use any name you wish. Using the standard variable D\$ in all

your programs makes them compatible with each other, and with everyone else's. Add a line like this to your BASIC programs:

```
10 D$ = "": REM  THERE IS AN INVISIBLE  
    CTRL-D BETWEEN THE QUOTES
```

or, in Applesoft, you may use:

```
10 D$ = CHR$ (4): REM  CHR$(4) = CTRL-D
```

Then wherever D\$ is used, it is easy to see that it is a CTRL-D character.

Try running the program shown below on your Apple II:

```
10 D$ = "": REM  CTRL-D  
20 FOR I = 1 TO 10  
30 PRINT D$; "CATALOG"  
40 NEXT I  
50 END
```

Unless you have more than 18 files on the diskette, you should see the catalog displayed 10 times, without touching a key. If there are more than 18 catalog entries, you'll have to press a key each time a pause occurs.

## USING DISK FILES

Apple DOS supports two types of disk files: *sequential* files and *random-access* files. Both types of files contain blocks of data, called fields. A field may be one character or many characters long.

### Sequential Files

Sequential files, as the name implies, can be accessed in a sequential manner only. To read or write the last field in the file, you must first read or write all previous fields. For some applications, sequential access is fine.

### Random-Access Files

Random-access files allow more flexibility than sequential files do. You can read or write any field in the file without regard to its location. For many applications, random-access files are the best solution.

## USING SEQUENTIAL FILES

In order to use sequential files, you will need to learn a few more DOS commands: OPEN, CLOSE, READ, and WRITE.

## Opening Sequential Files

Disk files must be *opened* before they can be accessed. Opening a disk file causes DOS to retrieve information about the file: whether it is on the disk, and if so, where it is on the disk. OPEN also sets aside an area of memory to be used as a *buffer* for the file. The buffer allows you to access a small portion of the file without activating the disk drive on every field access, and that saves a lot of time. The OPEN command looks like this:

```
OPEN FILENAME,S6,D2,V99
```

If the file does not exist as specified, DOS will create a new file entry in the disk directory.

Within a program the OPEN command must be in a PRINT statement and be preceded by a CTRL-D character.

The program below will do nothing more than create the sequential file SEQUENTIAL on the disk in the default drive:

```
100 D$ = "": REM CTRL-D
200 PRINT D$;"OPEN SEQUENTIAL"
300 END
```

We will call this *Program 1*.

You can include any combination of slot, drive, and volume parameters with the OPEN command:

```
100 D$ = "": REM CTRL-D
200 PRINT D$;"OPEN SEQUENTIAL,S6,D1,V123"
300 END
```

or:

```
200 PRINT D$;"OPEN SEQUENTIAL,V123,S6"
```

Note that all the parameters occur *before* the closing quotes, and that they are separated by commas.

After you run the program, the file SEQUENTIAL will be in the catalog. Confirm this by typing CATALOG. The catalog should look something like this:

```
DISK VOLUME 123

*A 002 HELLO
T 001 SEQUENTIAL
```

Note the letter T before the number of sectors in SEQUENTIAL. T is the code which designates SEQUENTIAL as a text file, as opposed to an Applesoft or Integer BASIC program file, or a binary file. The asterisk (\*) preceding the type of file HELLO indicates it has been locked.

## Closing Files

Actually, Program 1 is a very poor program, for one big reason: it does not close the file when it is finished. The CLOSE command is very important. Not closing files can result in loss of data, and possible destruction of data on another diskette

(see the section on soft crashes in this chapter). The CLOSE command has two formats. The first one is:

```
CLOSE
```

The CLOSE command with no parameters will close all open files, on all diskettes, regardless of which drive, slot, or volume they reside on.

Sometimes you may want to close one or more specific files. You close individual files by adding the file name to the CLOSE command, like this:

```
CLOSE FILENAME
```

No slot, drive, or volume parameters are allowed or required with either form of the CLOSE command. DOS knows where the file is, since the file is already open.

Program 1 should be corrected by adding this line:

```
290 PRINT D$;"CLOSE"
```

or, you could add:

```
290 PRINT D$;"CLOSE SEQUENTIAL"
```

### Writing to Sequential Files

Program 1 is a pretty useless program. Diskettes are used to store and retrieve information. Since you cannot retrieve anything that isn't already on the disk, we will discuss how to store information first.

Information is sent to the Disk II the same way it is sent to the screen or printer: via the PRINT statement. Anything you can print can be put in a disk file. In fact, you might visualize a sequential file as a TV screen, or even better, as paper in a printer.

When you print something on a file, DOS updates an internal pointer which points to the next location on the disk surface where data will be stored, just as a printer advances paper to the next line.

A sequential file pointer can only be moved forward. The OPEN command moves the pointer back to the beginning of the file.

Before you can print data to a disk file, you must first use a WRITE command to tell DOS that PRINT statements are to write to a file, instead of the display screen. For sequential files, the WRITE command looks like this:

```
WRITE FILENAME
```

After you issue the WRITE command, subsequent output will be directed to the specified file. Note that subsequent output will include any error messages. However, after the error message is stored on the file, the WRITE command is cancelled. You will see only the BASIC prompt and cursor on the display screen.

The WRITE command must be in a PRINT statement, preceded by a CTRL-D character. If you issue the WRITE command in immediate mode you will get the error NOT DIRECT COMMAND.

Add the following lines to Program 1:

```
210 PRINT D$;"WRITE SEQUENTIAL"  
220 PRINT "THIS TEXT WILL BE STORED IN THE FILE"
```

The program does the following:

1. It creates a file, if it needs to.
2. It opens the file.
3. It stores text in the file.
4. It closes the file.

You may insert as many PRINT statements between lines 210 and 290 as you wish. You may print text, numbers, and variables, in any combination, as long as the syntax for the PRINT statement is correct. For example:

```
220 FOR I = 1 TO 100  
230 PRINT I  
240 NEXT I  
250 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Note that you only use the CTRL-D character prefix with DOS commands, never when writing the contents of a file.

Be careful not to print characters to a file while FLASH or INVERSE statements are in effect, as these characters are not properly handled by DOS.

Remember that each time you run this program, whatever is in the PRINT statements will overwrite and erase data already stored on the file. If you PRINT fewer characters than are already in the file, the tail end of the previous data will remain, following the new data.

One way to circumvent the problem of leftover data is to erase the file before you store new data in it. The DELETE command (in the usual PRINT statement context) may be incorporated into the program just prior to the OPEN command. Every time the program is run the file will be deleted and then recreated by the OPEN command.

However, a problem will then occur if you try to run the program when there is no file called SEQUENTIAL on the diskette. If you do, you will see the error FILE NOT FOUND. You can prevent that by adding another OPEN command, just prior to the DELETE command. This is what happens:

1. The first OPEN command creates a file if one does not already exist.
2. The DELETE command erases the file, no matter when it was created.
3. The second OPEN command creates a new, empty file.

As you make these changes to Program 1, remember that once you have specified the slot, drive, and volume, you don't need to specify them again, unless you wish to refer to a different disk drive. With this in mind, use an OPEN command with the slot, drive, and volume parameters as the first DOS command, and allow subsequent commands to default to those values.

If you have made these changes to Program 1, your listing should now look like this:

```

100 D$ = "": REM CTRL-D
110 PRINT D$:"OPEN SEQUENTIAL,V123,S6,D1"
120 PRINT D$:"DELETE SEQUENTIAL"
200 PRINT D$:"OPEN SEQUENTIAL"
210 PRINT D$:"WRITE SEQUENTIAL"
220 PRINT "THIS TEXT WILL BE STORED IN THE FILE"
290 PRINT D$:"CLOSE"
300 END

```

Whenever you run this program, it will store the contents of the PRINT statements between lines 210 and 290 in the file SEQUENTIAL. You can use any file name you wish, but if you change a file name, be sure to also change every reference to the changed file name.

You can use a variable for the file name and have the program ask for the name of the file to access. With this modification, the program looks like this:

```

10 INPUT "FILE NAME: ";F$
100 D$ = "": REM CTRL-D
110 PRINT D$:"OPEN ";F$;",V123,S6,D1"
120 PRINT D$:"DELETE ";F$
200 PRINT D$:"OPEN F$"
210 PRINT D$:"WRITE ";F$
220 PRINT "THIS TEXT WILL BE STORED IN THE FILE"
290 PRINT D$:"CLOSE"
300 END

```

You can go even further and have the program request the slot, drive, and volume numbers by making these changes:

```

10 INPUT "FILE NAME: ";F$
20 INPUT "SLOT NUMBER: ";S
30 INPUT "DRIVE NUMBER: ";D
40 INPUT "VOLUME NUMBER: ";V
100 D$ = "": REM CTRL-D
110 PRINT D$:"OPEN ";F$;",S";S;",D";D;",V";V

```

When you make these changes, be sure to enter them exactly as shown. Do not forget to include the commas before the S, D, and V parameters, or DOS will not be able to distinguish the parameters from the file name.

To make this program really useful, it should let you input the text to be stored, instead of having to change the PRINT statements. One way to do this is:

```

150 INPUT "ENTER THE TEXT TO STORE: ";T$
.
.
.
220 PRINT T$

```

But that only allows you to enter and store one line of text. You could fix that by adding more INPUT statements, followed by more PRINT statements, but once again you have fixed the amount of text which can be entered and stored. Why not add a test after the INPUT statement so you can signal the end of your text?

Then add a GOTO after the PRINT so you can enter another line:

```

150 INPUT "ENTER THE TEXT TO STORE: ";T$
160 IF T$ = "END" THEN 290
210 PRINT D$;"WRITE ";F$
220 PRINT T$
230 GOTO 150

```

Now whenever you are finished entering text, you can type END and the file will be closed. But there is still one big problem. Remember that the WRITE command causes all output to be directed to the file. Since the INPUT statement outputs the prompt ENTER THE TEXT TO STORE:, that line will not appear on the screen after the WRITE has been executed; it will be stored as part of the diskette file.

The WRITE command must be cancelled in advance of output not intended for the diskette. Any DOS command will cancel the WRITE command, but the safest one to use is the *null command*, which is the CTRL-D character all by itself. Add this line to the program.

```

225 PRINT D$

```

With all these changes complete, you now have a program that will allow you to store any amount of text you wish (up to the maximum amount the diskette will hold), under any file name you choose, on any drive connected to your Apple II.

## Reading Sequential Files

Just as output can be directed to the disk, input can be accepted from a disk file. The READ command identifies a disk file as the source for data input. For sequential files the READ command looks like this:

```

READ FILENAME

```

The READ command must be in a PRINT statement, preceded by a CTRL-D character. If you issue the READ command in immediate mode you will get the error NOT DIRECT COMMAND.

After the READ command has been executed, subsequent INPUT statements receive data from the specified file until another DOS command, or an error, cancels the READ command. *Program 2*, below, demonstrates use of the READ command.

```

100 D$ = "": REM CTRL-D
110 INPUT "FILE NAME TO READ: ";F$
120 INPUT "SLOT NUMBER: ";S
130 INPUT "DRIVE NUMBER: ";D
140 INPUT "VOLUME NUMBER: ";V
150 PRINT D$;"OPEN ";F$;"S";S;"D";D;"V";V
160 PRINT D$;"READ ";F$
170 INPUT A$
180 PRINT A$
190 GOTO 170
200 END

```

Program 2 will display all of the text in a disk file created by Program 1. After all of the file's contents have been output, you will see the message END OF DATA and the program will stop with the message BREAK IN 170.



You may notice that Program 2 lacks a CLOSE command. We have explained why files must be closed when they are no longer needed, but in this case the file is never written to, therefore the directory or track/sector information never needs to be updated (assuming the file name you specify actually exists). The file should be closed, just to be safe.

It is not a good practice to write programs that print error messages and stop under circumstances as predictable as reaching the end of data in a file.

### Preventing the END OF DATA Error

The END OF DATA condition can be detected before an error is generated. Program 2 could branch control to a CLOSE command upon detecting an end of data. The easiest way to detect an END OF DATA is by using the Applesoft ONERR GOTO statement (discussed in Chapter 4), but ONERR GOTO is not available in Integer BASIC. (DOS error codes are explained in Appendix C.)

Another way to find the end of text in files written by Program 1 is to modify Program 1 so that when you type the word END all by itself, a special word or character is written to the file just before it is closed. Then change Program 2 to look for the special end of text marker and CLOSE the file properly. This method will work in both Integer BASIC and Applesoft.

### Integer BASIC and Applesoft Differences

The READ command identifies a disk file as the data source for subsequent INPUT statements. But the INPUT statement syntax must conform to the BASIC in which the program is written (see Chapter 8). Nevertheless, data supplied by a file depends on how that data was stored.

In Applesoft, for example:

```
100 D$ = "": REM CTRL-D
200 PRINT D$;"OPEN FILE1"
300 PRINT D$;"WRITE FILE1"
400 PRINT "HELLO","THIS IS A TEST"
500 PRINT D$;"CLOSE"
600 END
```

and the same program with this minor change:

```
400 PRINT "HELLO, THIS IS A TEST"
```

both store two lines of text on FILE1. Applesoft does not handle the comma as a separator the same way Integer BASIC does. If you try to read the text in FILE1 with:

```
300 PRINT D$;"READ FILE1"
400 INPUT A$
```

while in Applesoft, you will get the message ?EXTRA IGNORED and the variable A\$ will be set to HELLO, regardless of which line 400 was used to store that text.

In Applesoft, commas separate multiple values in a single PRINT statement.

Integer BASIC will accept both lines as one, so the value of A\$ stored by the first example will be HELLO THIS IS A TEST, and the value of A\$ from the second version of line 400 will be HELLO, THIS IS A TEST.

### Using the Applesoft GET Statement to Read Text Files

In Applesoft the GET statement can be used to read data from a disk file. The GET statement differs from the INPUT statement in that the GET statement returns one character at a time. Thus, if a file contains the text THIS FILE CONTAINS TEXT, and you execute an OPEN, a READ, and then a GET, the first GET will return the letter T. The next GET will return H, and subsequent GET statements will return I, S, a blank space, F, I, L, E and so on, until every character of the text has been read. If the GET returns a comma or a carriage return, the program can detect it and interpret it properly.

*Program 3*, shown below, demonstrates how a file may be read by using the Applesoft GET statement to build a line of text, one character at a time.

```

100 D$ = CHR$ (4): REM CTRL-D
200 INPUT "FILE NAME TO READ: ";F$
300 INPUT "SLOT NUMBER: ";S
400 INPUT "DRIVE NUMBER: ";D
500 INPUT "VOLUME NUMBER: ";V
600 PRINT D$;"OPEN ";F$;"",V"";V;"",D"";D;"",S"";S
700 PRINT D$;"READ ";F$
800 B$ = ""
900 GET A$
1000 IF A$ = CHR$ (13) THEN 1300
1100 B$ = B$ + A$
1200 GOTO 900
1300 REM RETURN CHARACTER FOUND
1400 REM B$ IS COMPLETE
1500 PRINT B$
1600 GOTO 800
1700 END

```

But the GET statement has a problem when used with disk files. The first character printed after a GET has been executed is ignored. If the first character printed is a DOS command, then the CTRL-D character will be ignored, which means the entire command will be printed, not executed as a DOS command.

The cure for this problem is to print a *throw-away* character first, one that is intended to be ignored. A good character to use is CTRL-A (ASCII code 1), since it is nonprinting and has no special meaning (as opposed to CTRL-D, for example).

Program 3 can be *patched*, or corrected, by changing line 1500 to read:

```

1500 PRINT CHR$ (1);B$: REM CHR$(1) = CTRL-A

```

### Storing Numbers in Files

You may have used Program 1 to store numbers in a file, either as part of some text, as in:

```
220 PRINT "MY ADDRESS IS 1234 NORTH STREET"
```

or directly as numeric values:

```
230 PRINT 1,2,3,4,5
```

or via numeric variables:

```
240 PRINT A,B,C,D,E
```

If you stored numbers directly, you will get some strange results when you READ them back with Program 2.

Using Integer BASIC, numbers separated by commas or semicolons will form one big number (12345 from line 230 above), for a total of one line of output.

Using Applesoft, things become even more confusing. The first three numbers are concatenated (123) and printed as one line of output. The next two values (4 and 5) produce one line of output each, for a total of three lines of output.

These problems result from the format used to store information in disk files. Commas are not stored between the numbers if they would not be displayed on the screen. Instead, on the screen the commas cause the next value to appear at the next TAB position. When output is directed to a disk file, however, DOS discards commas completely. Nothing equivalent to tabbing occurs. As a result, values become concatenated until a carriage return (ASCII code 13) separates them. The carriage return is the only character DOS interprets as a value separator. Integer BASIC outputs a carriage return after the fifth tab stop (comma), while Applesoft does it after three tab stops.

To avoid problems, you should make sure that each numeric field you store in a disk file is followed by a carriage return character. The easiest way to do this is to output each number with a separate PRINT statement:

```
230 PRINT 1: PRINT 2: PRINT 3: PRINT 4:
    PRINT 5
```

Another method available to Applesoft users is to include a carriage return character as a separator in the PRINT statement parameter list:

```
230 PRINT 1;CHR$(13);2;CHR$(13);3;CHR$(1
    3);4;CHR$(13);5
```

You may prefer to define a variable as a carriage return character, then print the variable:

```
11 R$ = CHR$(13): REM RETURN CHARACTER
    .
    .
    .
230 PRINT 1;R$;2;R$;3;R$;4;R$;5
```

This makes for a cleaner program.

## HOW TO APPEND TO SEQUENTIAL FILES

A sequential file should be closed when written into. When you CLOSE a file, you lose track of where the last item was stored. To add data to the end of the file, therefore, you must first *find* the end of the file. You could read each item in the file until you reach the last one, but that can be very time consuming with large files. The APPEND command does the work for you.

The APPEND command places the file pointer at the first unused character beyond the end of the file. If you read after the APPEND command has been issued, you'll get an END OF DATA error. If you write after an APPEND, the new data will be added to the end of the data already on file.

The APPEND command is used in place of the OPEN command. There are two important differences between the APPEND and OPEN commands:

1. APPEND requires that the file already exist. If it does not, the error FILE NOT FOUND is returned. APPEND will not create a file; APPEND assumes the file exists.
2. APPEND places the pointer at the *end* of the file. OPEN places the pointer at the *beginning* of the file.

The format for the APPEND command is the same as for the OPEN command. Here is an example:

```
APPEND FILENAME,S6,D2,V99
```

As always, the slot, drive, and volume commands are optional.

## THE POSITION COMMAND

Another useful command is the POSITION command. POSITION moves the pointer *forward* (never backward) by the specified number of fields relative to the current pointer position. POSITION looks like this:

```
POSITION FILENAME,R30
```

R indicates relative field; the number after R is the number of fields to skip. Fields are marked by carriage return characters, so the POSITION command above counts 30 carriage returns beyond the current position and moves the file pointer there. If you specify R0, the pointer is not moved.

POSITION actually examines the file character-by-character, starting from the current position. If there are not enough fields, or if an unused byte is encountered, the error END OF DATA is displayed immediately. Executing an INPUT or a GET is not necessary to cause the error.

A file must be open before it can be referenced by a POSITION command. When you open a file, the pointer is set to the beginning of the file. If you issue a POSITION command when the pointer is at the beginning of the file, it will effectively select the *absolute* field within the file.

Remember, just like any other DOS command, POSITION cancels both the READ and WRITE commands. Be sure to execute POSITION before issuing a READ or WRITE command, not after.

## USING RANDOM-ACCESS FILES

Random-access files are structured into sections called *records*. Each record in a particular file holds the same amount of information, which is defined as some number of bytes (characters) when the file is created.

The amount of information which can be stored in one record is referred to as the record *length*.

Records are identified by a number indicating their absolute position in the file. The first record in every file is record number 0, the next is number 1, followed by record 2, and so on.

The smallest random-access file has one record. Files expand as new records are added, but they do not shrink. To remove unwanted records from a random-access file and shrink the size of the file, you must copy the records which are to be preserved into a new random-access file.

Programs must specify which record of a random-access file is selected, and what part of the record is to be accessed.

### Opening Random-Access Files

To define a file as a random-access file, you must include an additional parameter when the file is opened: the *length* parameter. The length parameter (L) specifies the length of each record, like this:

```
OPEN FILENAME,L10,S6,D1,V100
```

The length parameter must have a value ranging between 1 and 32767. It does not have to be the first parameter in the list, but it must be present if the file is to be random-access.

Programs should never write records that are longer than the number of bytes specified by the length parameter. That includes carriage returns and commas. If too many characters are stored in a record, the succeeding record may be overwritten or combined, creating a real mess.

### Closing Random-Access Files

The CLOSE command is identical for both sequential and random-access files.

### Random-Access Read and Write

The READ and WRITE commands require a *record* parameter for random-access files. The record parameter moves the file pointer to the beginning of a record. The following example uses the record parameter (R):

```
      READ FILENAME,R13  
or:   WRITE FILENAME,R6
```

The record parameter need not be the only one in the list. You may also specify slot, drive, and volume. Parameters may appear in any order. If the record parameter is not present, the pointer is not moved.

## A PRACTICAL RANDOM-ACCESS EXAMPLE

The following programs demonstrate a practical use of random-access files. The programs will work in either Applesoft or Integer BASIC. In order to use the second program in Integer BASIC, you must add a DIM statement before line 100 for variables B\$ and C\$ (255 characters each). The first program creates a file called RANDOM.

```

10 REM          RANDOM-ACCESS FILE CREATING PROGRAM
20 REM
30 REM          BE SURE TO RUN THIS PROGRAM FIRST AS IT
40 REM  STORES INFORMATION IN RECORD ZERO WHICH MUST
50 REM  EXIST FOR THE NEXT PROGRAM TO FUNCTION.
60 REM
100 D$="" : REM  CTRL-D
200 PRINT D$:"OPEN RANDOM,S6,D1,L256" : REM  OPEN THE FILE
300 PRINT D$:"WRITE RANDOM,R0" : REM  WRITE, RECORD ZERO
400 PRINT 0 : REM  STORE A ZERO IN RECORD ZERO
500 PRINT D$:"CLOSE" : REM  CLOSE THE FILE
600 END

```

Once the file has been created, the second program allows you to read, change, add, and list the records in the file. Each record will contain one line of information which you type in at the keyboard.

```

10 REM          RANDOM-ACCESS FILE DEMONSTRATION PROGRAM
20 REM
30 REM          THIS PROGRAM WILL MAINTAIN A RANDOM-ACCESS
40 REM  FILE WHICH CONSISTS OF SINGLE LINE RECORDS.
50 REM
60 REM          RECORD ZERO CONTAINS A NUMBER INDICATING
70 REM  THE LAST RECORD NUMBER IN USE.
80 REM
85 REM  NOTE:  FILE "RANDOM" MUST BE CREATED BEFORE
90 REM          ATTEMPTING TO USE THIS PROGRAM.
95 REM
100 D$="" : REM  CTRL-D
200 PRINT D$:"OPEN RANDOM,S6,D1,L256"
225 PRINT D$:"READ RANDOM,R0" : REM  READ RECORD ZERO
250 INPUT M : REM  M = THE LAST RECORD NUMBER IN USE
275 PRINT D$ : REM  CANCEL READ COMMAND
300 CALL -936 : REM  CLEAR SCREEN
400 PRINT "RANDOM-ACCESS FILE DEMONSTRATION"
500 PRINT : PRINT : PRINT : PRINT
600 PRINT "COMMANDS:" : PRINT
700 PRINT " 0 = STOP"
800 PRINT " 1 = READ A RECORD"
900 PRINT " 2 = ADD A RECORD"
1000 PRINT " 3 = CHANGE A RECORD"
1100 PRINT " 4 = LIST ALL RECORDS"
1200 PRINT : PRINT
1300 PRINT "WHICH";
1400 INPUT C
1500 IF C=0 THEN 8300 : REM  BRANCH
1600 IF C=1 THEN 2200 : REM  TO
1700 IF C=2 THEN 3300 : REM  THE
1800 IF C=3 THEN 4600 : REM  SELECTED

```

```

1900 IF C=4 THEN 6700 : REM  ROUTINE
2000 GOTO 300 : REM  (OR RE-DISPLAY THE MENU)
2050 REM
2100 REM  * * * * *  READ A RECORD  * * * * *
2150 REM
2200 CALL -936 : REM  CLEAR SCREEN
2300 PRINT : PRINT "READ A RECORD" : PRINT
2400 PRINT "WHICH RECORD NUMBER (0 TO STOP)";
2500 INPUT R
2600 IF R<1 THEN 300 : REM  RETURN TO MAIN MENU
2650 IF R>M THEN 2200 : REM  RECORD DOES NOT EXIST
2700 PRINT D$;"READ RANDOM,R";R : REM  PREPARE TO READ RECORD
2800 INPUT B$ : REM  READ THE DATA
2900 PRINT D$ : REM  CANCEL READ COMMAND
3000 PRINT : PRINT B$ : PRINT : REM  DISPLAY THE DATA
3100 GOTO 2400 : REM  ASK FOR ANOTHER RECORD NUMBER
3150 REM
3200 REM  * * * * *  ADD A RECORD  * * * * *
3250 REM
3300 CALL -936 : REM  CLEAR SCREEN
3400 PRINT : PRINT "ADD A RECORD" : PRINT
3500 PRINT "NEXT RECORD NUMBER = ";M+1
3600 PRINT : PRINT "ENTER DATA FOR RECORD ";M+1
3625 PRINT "(PRESS [RETURN] NOW TO STOP ADDING)"
3700 INPUT B$ : REM  GET USER'S RESPONSE
3750 IF B$ = "" THEN 300 : REM  QUIT IS JUST [RETURN]
3800 PRINT D$;"WRITE RANDOM,R";M+1 : REM  PREPARE TO WRITE
3900 PRINT B$ : REM  SEND DATA TO FILE
4000 M = M + 1 : REM  INCREMENT LAST RECORD NUMBER
4100 PRINT "WRITE RANDOM,R0" : REM  PREPARE TO WRITE RECORD
    ZERO
4200 PRINT M : REM  STORE UPDATED VALUE
4300 PRINT D$ : REM  CANCEL WRITE COMMAND
4400 GOTO 3500 : REM  LOOP FOR ANOTHER RECORD
4450 REM
4500 REM  * * * * *  CHANGE A RECORD  * * * * *
4550 REM
4600 CALL -936 : REM  CLEAR SCREEN
4700 PRINT : PRINT "CHANGE A RECORD" : PRINT
4800 PRINT "CHANGE WHICH RECORD (0 TO STOP)";
4900 INPUT R
5000 IF R<1 THEN 300 : REM  RETURN TO MAIN MENU
5050 IF R>M THEN 4600 : REM  TRY AGAIN IF RECORD NOT ON FILE
5100 PRINT D$;"READ RANDOM,R";R : REM  PREPARE TO READ
5200 INPUT B$ : REM  READ THE RECORD
5300 PRINT D$ : REM  CANCEL READ COMMAND
5400 PRINT : PRINT B$ : PRINT : REM  DISPLAY THE DATA
5500 PRINT "ENTER THE NEW DATA"
5600 PRINT "(PRESS [RETURN] NOW TO CANCEL CHANGES)"
5700 PRINT
5800 INPUT C$ : REM  GET USER'S RESPONSE
5900 IF C$>"" THEN 6200 : REM  BRANCH IF NEW DATA
6000 PRINT "RECORD ";R;" UNCHANGED ! ! !" : REM  LOOP IF
6100 GOTO 4800 : REM  NO CHANGES DESIRED
6200 PRINT D$;"WRITE RANDOM,R";R : REM  PREPARE TO WRITE
6300 PRINT C$ : REM  STORE CHANGED DATA

```

```

6400 PRINT D$ : REM  CANCEL WRITE COMMAND
6500 GOTO 4800 : REM  LOOP FOR ANOTHER RECORD TO CHANGE
6550 REM
6600 REM  * * * * *      LIST ALL RECORDS      * * * * *
6650 REM
6700 CALL -936 : REM  CLEAR SCREEN
6800 PRINT : PRINT "LIST ALL RECORDS" : PRINT
6900 R = 0 : REM  RESET THE COUNTER
7000 R = R + 1 : REM  INCREMENT THE COUNTER
7100 IF R > M THEN 7700 : REM  STOP AFTER LAST RECORD
7200 PRINT D$;"READ RANDO ,R";R : REM  PREPARE TO READ
7300 INPUT B$ : REM  READ THE DATA
7400 PRINT "RECORD NUMBER : ";R : REM  DISPLAY RECORD NUMBER
7500 PRINT B$ : PRINT : REM  DISPLAY RECORD'S DATA
7600 GOTO 7000 : REM  LOOP FOR NEXT RECORD
7700 PRINT D$ : REM  CANCEL READ COMMAND
7800 PRINT : PRINT "* * * * * END-OF-FILE" : REM  PRINT
    EOF MESSAGE
7900 PRINT "PRESS RETURN TO CONTINUE"; : REM  REQUEST
    RESPONSE
8000 INPUT B$ : REM  GET USER'S RESPONSE
8100 GOTO 300 : REM  RETURN TO MAIN MENU
8150 REM
8200 REM  * * * * *      STOP PROGRAM      * * * * *
8250 REM
8300 PRINT D$;"CLOSE" : REM  CLOSE THE FILE
8400 CALL -936 : REM  CLEAR SCREEN
8500 FOR I=1 TO 24 : PRINT : NEXT I : REM  MOVE TO BOTTOM
    LINE
8600 PRINT "PROGRAM COMPLETE."
8700 END

```

## THE BYTE PARAMETER

*Byte* is another useful random-access file parameter. The byte parameter is used with the READ, WRITE, and POSITION commands to move the pointer to the specified byte (character) within any selected record. A comma and the letter B may be added to a READ, WRITE, or POSITION command. The record parameter must be present with the byte parameter. Here is an example:

```
READ FILENAME,R19,B3
```

In this example, reading will begin at the fourth byte in the 20th record. (Remember, the first byte is byte number 0.) The byte parameter can move the pointer backward or forward within the record.

If you are going to use the byte parameter, records must have an exact data format. You must know the byte position of each field within the record, or you will probably end up with meaningless data.

When using the byte parameter with the POSITION command, remember that POSITION cancels any prior READ or WRITE command. You could execute a READ or WRITE command that includes a record parameter, then use POSITION to move forward to the correct field. But you must then execute another READ or



WRITE, since POSITION cancels the prior READ or WRITE command.

POSITION will only move the pointer forward to another field in the current record. Within that field, the byte parameter allows you to reference subfields, one character at a time.

## OTHER DOS COMMANDS

There are a few commands which have not been explained yet. These commands are EXEC, MAXFILES, TRACE, MON, and NOMON.

### EXEC

EXEC is a very special command. EXEC allows you to turn control of your Apple II over to a text file. The EXEC command looks like this:

```
EXEC FILENAME,R6,S5,D2,V23
```

The R parameter is similar to the POSITION command's record parameter. It refers to the relative field number of the EXEC file where processing should begin. Since EXEC opens the file and places the pointer at the beginning of the EXEC file, the R parameter refers to an absolute field number within the file. R0 is the default condition, which starts execution at the beginning of the file. R1 starts execution at the second field.

The relative record, slot, drive, and volume specifications are all optional and can occur in any order.

When EXEC is issued, the file specified is opened, then implicit READ and INPUT statements take place. The first line in the file is retrieved if no R parameter is present. If R is present, then the line specified by R is retrieved.

The retrieved line is treated as if it had been typed on the keyboard in immediate mode. If the line is meaningless garbage, you will see the message ?SYNTAX ERROR or \*\*\* SYNTAX ERR. If it is a valid program line, like:

```
100 PRINT "THIS IS A TEST"
```

it will be stored in memory, as if you had just typed it. If it is a direct command, such as LIST or RUN, it will be executed.

After action occasioned by the first line is completed, the next line is read and acted upon. This continues until the end of the file has been reached; at that time control of the Apple II returns to the keyboard.

Suppose a text file called BUBBA contains these lines of text:

```
PRINT "I HAVE CONTROL OF YOUR APPLE ! ! !"
FP
100 GR
200 FOR I=0 TO 39
300 FOR J=0 TO 39
400 COLOR=RND(0)*15
```

```

500 PLOT I,J
600 NEXT J
700 NEXT I
800 FOR I=1 TO 5000 : NEXT I
900 TEXT : CALL -936
999 END
LIST
FOR I=1 TO 5000 : NEXT I
RUN
NEW
PRINT "FINISHED.  HERE'S YOUR DISK CATALOG:"
CATALOG

```

When you type the command:

```
EXEC BUBBA
```

several things will happen. First a message will be displayed. Then Applesoft will be invoked. Next, a short program is entered into memory and listed. After a brief pause the program is RUN, then erased from memory. Another message is displayed, followed by the catalog of the currently selected disk.

### Notes on EXEC

EXEC has several interesting kinks you should know about.

Only one EXEC file may be open at any one time. If a file which is being accessed by an EXEC command contains an EXEC command, the first file is closed and the second file takes control.

After the EXEC file runs a program, the next EXEC file field (i.e., line) is executed. If the program is aborted with CTRL-C, the EXEC will usually not continue.

If a program run by the EXEC command contains an INPUT statement, the input is taken from the next field in the EXEC file. That causes a problem if the next field is an immediate mode DOS command (not a program line). The command will be executed instead of being accepted as INPUT data.

If a program executes a CLOSE command, or if the EXEC file contains an immediate mode CLOSE command, the EXEC file will not be closed.

### Using EXEC to Convert a Program from One BASIC to the Other

EXEC can be used to convert programs between Integer BASIC and Applesoft. First store the program as a text file. You can do this by adding a few lines to the program to create the text file, execute a WRITE command, then list the program. The listing goes into the text file. Here are the necessary statements:

```

1 D$ = "": PRINT D$;"OPEN FILE": PRINT D$;"WRITE FILE"
2 LIST 10,32767
3 PRINT D$;"CLOSE": END
10 REM YOUR PROGRAM BEGINS HERE

```

When you run the program, only lines 1, 2, and 3 are executed. They will open a file and store the program listing in it. When this finishes, you can change to the other BASIC (with FP or INT, for example). Then type:

EXEC FILE

The old program lines will be loaded from the disk into memory. Now you must edit the program to correct the parts that don't conform to the rules of the new language.

## MAXFILES

MAXFILES allows you to specify the maximum number of files which may be open at any one time. Each file has 595 bytes of memory reserved for its use as a buffer. There are two 256-byte sections in each buffer, one for reading and the other for writing. The remaining 83 bytes are used for housekeeping information, such as the track/sector list.

When a file is opened, or a disk operation such as CATALOG or LOCK occurs, DOS reads 256 bytes of information from the disk and places it in a buffer. If the information has been changed and needs to be written back to the diskette, 256 bytes are copied from the buffer and stored on the diskette.

The following example specifies eight files maximum:

MAXFILES 8

The number of files must be an integer from 1 to 16. When DOS is booted, three buffers are allocated. MAXFILES may be set higher if you intend to use more than three files simultaneously. MAXFILES may be set lower if you need those extra bytes of memory for your application.

One buffer is used just to execute any of the following DOS commands:

APPEND	FP	POSITION
BLOAD	INIT	READ
BRUN	INT	RENAME
BSAVE	LOAD	RUN
CATALOG	LOCK	SAVE
CHAIN	MAXFILES	UNLOCK
CLOSE	MON	VERIFY
DELETE	NOMON	WRITE
EXEC	OPEN	

Thus, if you have opened disk files up to the limit and then issue the CATALOG command, the error NO BUFFERS AVAILABLE is returned. No buffer is required for commands used outside of the DOS context, however (e.g., cassette LOAD).

When MAXFILES is changed, Integer BASIC programs are erased and Applesoft strings become garbled. Therefore, execute MAXFILES before loading or running a program.

MAXFILES may be executed within Applesoft programs if preceded by a CTRL-D character in a PRINT statement. MAXFILES will cause GOTO, GOSUB, and other

instructions to malfunction unless it is the first statement in the program. In order to avoid having strings clobbered, use MAXFILES as follows:

```

11 REM FIRST ISSUE MAXFILES COMMAND
12 PRINT CHR$(4); "MAXFILES 9"
13 REM REGULAR PROGRAM BEGINS HERE
14 D$=CHR$(4):REM CTRL-D CHARACTER

```

You can only use MAXFILES in Integer BASIC in the context of an EXEC file. For example, the EXEC file contains three psuedo-immediate mode statements:

```

MAXFILES 8
LOAD PROGRAM
RUN 10

```

It is invoked in a program as follows:

```

5 PRINT D$;"EXEC MXF":REM SET MAXFILES
  CONTINUE AT LINE 10
10 REM PROGRAM BEGINS HERE

```

## USING DOS DEBUGGING AIDS

Since programs which access disk files are usually somewhat complex, DOS has three commands that help you debug programs: MON, NOMON and TRACE.

### MON

MON is short for monitor. MON allows you to monitor the information going to and coming from the disk. MON uses three parameters. The following example shows all three:

```
MON C,I,O
```

MON parameters specify the type of information to be displayed. C causes commands to the disk to be monitored. I causes input from the disk to be displayed. O causes output to the disk to be displayed.

The parameters may appear in any order, and in any combination. At least one parameter must be present or the command will be ignored.

MON will remain in effect until a NOMON, INT, or FP command is executed, or DOS is rebooted.

### NOMON

NOMON cancels the effect of the MON command. NOMON uses the same three parameters MON uses, but the NOMON parameters specify which data is not to

be monitored. For example, assuming MON I,O,C has been issued, the command:

```
NOMON 0
```

will cancel monitoring of output to the disk. Input from the disk and DOS Commands will continue to be displayed.

### Using the TRACE Command

The TRACE command (see Chapter 4) is used when debugging programs. But since TRACE prints line numbers with no carriage return, DOS commands become concatenated to the line number, and ignored. The remedy for this is simple. A carriage return should be output prior to each DOS command. To do this, change the line:

```
100 D$ = CHR$ (4): REM CTRL-D
```

to read:

```
100 D$ = CHR$ (13) + CHR$ (4): REM RETURN + CTRL-
```

Then, whenever PRINT D\$ occurs, the CTRL-D character will be preceded by a carriage return, separating it from the traced line numbers.

Another problem exists, though. If a WRITE command is in effect, all those line numbers will be stored in the file. (Sorry, no way around this one.)

## MACHINE LANGUAGE (BINARY IMAGE) DISK FILES

The Disk II supports machine language and binary image (graphics) files. These files are shown with the letter B as the file type code in a disk catalog.

Both low-resolution and high-resolution images can be stored on diskettes for later recall and display.

Machine language programs can be loaded and executed directly, or they may be called by BASIC programs using the CALL statement or USR function.

DOS has three commands which are specifically designed for binary files. They are BSAVE, BLOAD, and BRUN. The effect of each command is the same as its nonbinary equivalent (BSAVE=SAVE, BLOAD=LOAD, BRUN=RUN).

### BSAVE

BSAVE, as the name implies, saves a binary image on the disk. Here is an example:

```
BSAVE FILENAME, A378, L21, S6, D2, V6
```

Note that there are two parameters not found on other DOS commands. Note also that these parameters are *not* optional; they must be specified. The slot, drive, and volume parameters are optional, as usual.

Parameter A is the *address* parameter; it refers to the starting memory address of the binary image to be saved. The address may be either decimal or hexadecimal constants. Hexadecimal values must be preceded by a dollar sign (\$). Decimal values must be in the range 0 to 65535. Negative values are prohibited.

The L parameter specifies the *length* of the binary image to be saved. The length is the number of bytes in the image. It may be specified as a decimal or hexadecimal number, with hexadecimal values preceded by a dollar sign (\$). The length must be in the range 1 through 32767 or a SYNTAX ERROR will be generated. 32767 bytes is the largest size DOS can store as a single field. Two BSAVE statements must be used to store more than 32767 bytes.

If the memory locations in the range specified do not physically exist, no error will be returned. It is not very useful to specify a location outside of the installed range on your Apple II (e.g., 49151 or \$BFFF on a 48K system).

## BLOAD

BLOAD retrieves the contents of binary files and loads them into memory. The BLOAD command looks like this:

```
BLOAD FILENAME, A378, L21, S6, D2, V6
```

The address parameter is optional with the BLOAD command. If it is absent, the image will be loaded beginning at the address specified when the image was saved.

Machine language programs may not function properly if they are loaded into the wrong memory addresses.

Unlike the LOAD command, BLOAD will not erase programs or data values unless they reside in the memory locations where the image will be stored. Only those locations within the BLOAD range are affected; no other memory values are changed.

No error will be returned if you specify read-only memory (ROM) locations as part of the BLOAD range. The ROM locations will be unchanged, of course.

## BRUN

BRUN is identical to BLOAD except that after the file has been loaded, BRUN executes a machine language JMP (jump) instruction to the starting address. If no address was specified, the jump will be to the address from which the image was saved. The following is a BRUN example:

```
BRUN FILENAME, A378, L21, S6, D2, V6
```

Never use BRUN with a graphic image, as the results are unpredictable.

# 6

## Graphics and Sound

The Apple II has capabilities for color video graphics and sound generation. Together, these features add another dimension to the programs you use, as well as those you might write yourself. This chapter is suited for the novice who may have just learned BASIC (perhaps by reading this book), as well as the assembly language programmer. Graphics are not difficult to master, especially in a high-level language. The Apple II Monitor, with its built-in machine language subroutines, helps the assembly language programmer use graphics and the Apple II onboard speaker even more. Once you finish this chapter you will have enough solid working knowledge of these features to start using them in programs you write.

### LOW-RESOLUTION GRAPHICS

The Apple II has two separate areas of memory available for low-resolution graphics. These two areas are called *pages*. Either low-resolution page can appear on the display screen as a graphics display 40 columns across by 48 rows, as shown in Figure 6-1.

Each coordinate (intersection of a row and column) appears as a small rectangle on the display screen. Each page contains 1,920 coordinates (40 columns times 48 rows), and you can assign any one of 16 colors to each coordinate on a

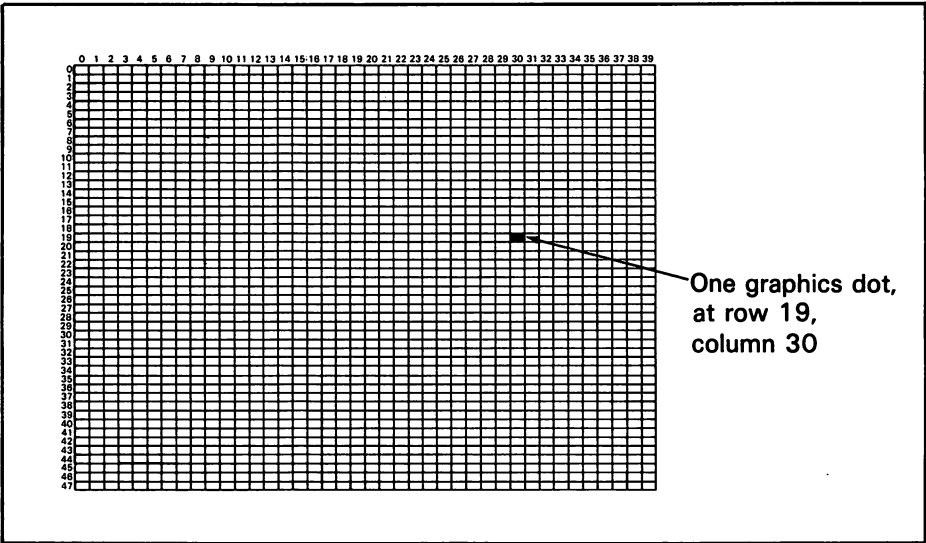


FIGURE 6-1. Low-Resolution Graphics Screen

page. Table 6-1 shows the hues which are available. You don't have to know the inner workings of the Apple II in order to use low-resolution graphics; a working knowledge of programming in BASIC (plus a little coordinate plotting) is sufficient to get you started.

**SETTING UP THE GRAPHICS PAGE**

The page dedicated to low-resolution graphics also doubles as the text page for the Apple II. When using BASIC, you switch to graphics mode from text mode by using the statement:

GR

Once this statement executes, the display screen goes black except for four lines at the bottom of the screen which hold text. This lower area of the screen is

TABLE 6-1. Low-Resolution Graphics Colors

Color	Number	Color	Number
Black	0	Brown	8
Magenta	1	Orange	9
Dark Blue	2	Gray #2	10
Purple	3	Pink	11
Dark Green	4	Light Green	12
Gray #1	5	Yellow	13
Medium Blue	6	Aqua	14
Light Blue	7	White	15



called the *text window*. With the text window at the bottom of the screen, space exists for 40 of the 48 rows available in low-resolution graphics mode. You can use this statement more than once in a program, even when you are in low-resolution graphics mode, as a means of clearing the screen.

### Full-Screen Graphics

After executing the GR statement, you can eliminate the text window to allow the last eight lines of graphics by entering:

```
POKE -16302,0
```

The text window disappears, replaced by graphics.

### Restoring the Text Window

If the Apple II is in full-screen graphics mode, you can restore the text window in two ways. To clear the graphics screen and restore the text window at the same time, use the GR statement. If you want to restore the text window without altering the first 40 rows of graphics, enter the statement:

```
POKE -16301,0
```

Once executed, this statement reopens the text window at the bottom of the screen.

### Going Back to Full-Screen Text

To leave the low-resolution graphics mode and return to the full-screen text mode, this BASIC statement:

```
TEXT
```

resets the display from graphics to characters. While the GR statement clears the screen when it executes, TEXT does not. Remember that text and graphics both use the same area of memory. Once this statement executes, you will probably see a screen full of odd characters; this is because the Apple II is now interpreting graphics data in memory as text. Clear the text screen with the Esc-@ key sequence, with the CALL -936 BASIC statement, or in Applesoft, with the HOME command.

## GRAPHICS PROGRAMMING STATEMENTS

Both Integer BASIC and Applesoft recognize low-resolution graphics commands to plot single coordinates on the screen, change the color used for plotting, and draw vertical or horizontal lines of varying lengths. These commands work only on low-resolution graphics page 1 (also called the primary page). If you use page 2, you eliminate many of these timesaving statements and must resort to such non-descriptive statements as PEEK, POKE and CALL.

### The COLOR Statement

In Table 6-1, each color listed has a corresponding number from 0 to 15. It is this number which you use in a COLOR statement to set the current low-resolution hue. For example:

```
COLOR=13
```

sets the drawing color to yellow. If you neglect to select a color, the Apple II chooses black, equivalent to COLOR=0, as the default color. Although you can specify a color number as high as 255 without generating a syntax error, COLOR only pays attention to the low-order *nybble* of the color number you select. Therefore, if you enter COLOR=222 (equal to DE hexadecimal), it will evaluate to COLOR=15 (equal to 0E hexadecimal).

### The PLOT Statement

This BASIC statement places a single graphics dot — actually a small rectangle — on the Apple II display screen at the coordinates you specify.

The statement:

```
PLOT 23,18
```

illuminates the graphics point at the 24th row and 19th column in the hue selected by the latest COLOR statement executed. This row number ranges from 0 to 47, and the column value from 0 to 39. If you exceed these limits in a PLOT statement, you will get an error message, and your program will stop. As with any low-resolution graphics statement (except GR), you can replace literal expressions with variables:

```
PLOT Y/2+12,X-4
```

### A Plotting Example

The following Integer BASIC program uses all of the low-resolution graphics statements discussed so far in this chapter. The object is to plot a diagonal line from the upper lefthand to lower righthand corners of the screen.

```
10 REM  DRAW A DIAGONAL LINE
11 REM  ACROSS THE LOW-RES SCREEN
20 GR
30 COLOR= RND (16)
40 FOR Y=0 TO 39
50 PLOT Y,Y
60 NEXT Y
70 GOTO 30
```

The display screen will look like Figure 6-2, except that the diagonal line changes colors randomly.

To convert this program to Applesoft, change line 30 as shown below.

```
30 COLOR= RND (16) * 16
```

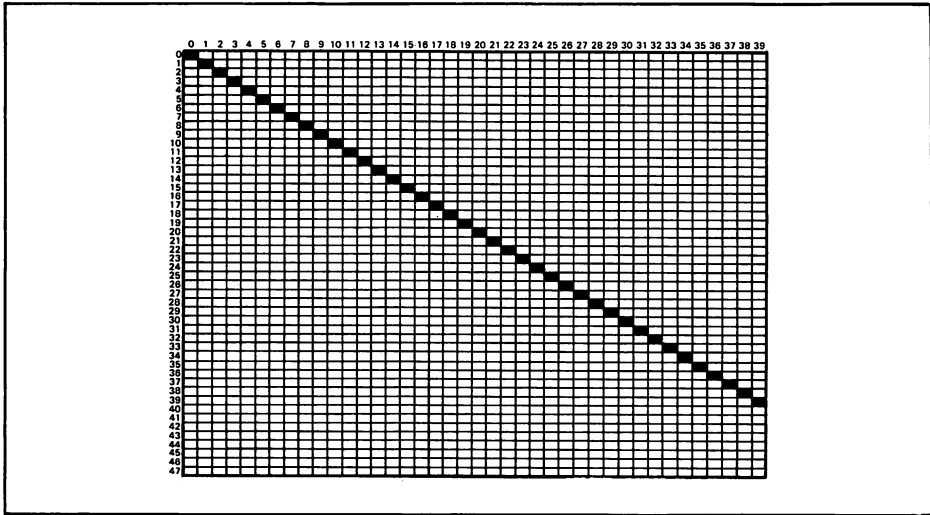


FIGURE 6-2. Low-Resolution Plotting Example

### Drawing Horizontal Lines

The HLIN command allows you to draw lines of varying lengths from left to right on the low-resolution graphics page. This statement:

```
HLIN 0,39 AT 0
```

draws a horizontal line at the extreme top of the screen, from the left margin to the right margin. HLIN stands for horizontal line. The general format of this statement is to enter the lefthand column followed by a comma, the righthand column to draw to (39 in this case), the word AT, and the row on which to draw the line.

The lefthand and righthand column parameters cannot be negative and must be less than 256; the righthand column cannot be smaller than the lefthand column parameter. The parameter following AT cannot be negative or larger than 48. If any of the parameters exceed the limits, an error message appears, stopping execution. If you specify column numbers greater than 39 in the HLIN statement in Integer BASIC, you will receive unpredictable results. Applesoft gives you an error message. For example, execute the statements:

```
10 GR
20 COLOR=12
30 HLIN 45,100 AT 0
```

In Integer BASIC, they cause two green bars to display. The first bar is not on row 0 (the highest line on the display) as it should be, and the bar continues several lines down on the screen. It is rarely a good idea to let these values exceed actual locations on the low-resolution screen.

## Drawing Vertical Lines

The VLIN (for vertical line) statement draws a line in the selected color, from one row to another in a specified column.

For example:

```
VLIN 12,30 AT 33
```

draws a line from row 12 down to row 30 at column 33. The parameter values for VLIN are identical to those for HLIN: the first and second expressions must be integers between 0 and 255 inclusive, the second parameter cannot be less than the first, and the last parameter (corresponding to the column number) must be between 0 and 39 inclusive. If any of these parameters is out of the ranges allowed, an error message displays on the Apple II screen.

## Using HLIN and VLIN in a Program

Another program (listed below) helps explain the use of HLIN and VLIN. Here, random lines draw in random colors on the screen. To stop the program, press CTRL-C.

```
10 REM LOW-RES HLIN AND VLIN DEMO
20 GR : REM USE GRAPHICS PAGE 1
30 POKE -16302,0: REM SET FULL-PAGE
40 CALL -1998: REM CLEAR ALL 48 ROWS
50 REM BEGIN PROGRAM
60 COLOR= RND (16): REM SELECT RANDOM COLOR
70 HLIN 0, RND (40) AT RND (48)
80 COLOR= RND (16)
90 VLIN 0, RND (48) AT RND (40)
100 GOTO 60
```

## The SCRN Statement

The SCRN statement is a bit more subtle than the other low-resolution graphics statements. Suppose you want the computer to figure out what color is displayed at a certain point on the screen. SCRN does this. This statement:

```
X=SCRN(12,24)
```

assigns the color number of the coordinates in parentheses (in this case, the 13th row and 25th column) to the variable X. The color passed back to the variable is numbered 0 through 15; the number corresponds to one of the low-resolution colors shown in Table 6-1. For example, if you enter the following immediate mode statements:

```
GR
COLOR=14
PLOT 12,12
PRINT SCRN (12,12)
```

the Apple II responds:

14

This statement may be very useful to you when you write advanced low-resolution graphics programs.

## HIGH-RESOLUTION GRAPHICS

The most fetching aspect of the Apple II computer is its high-resolution graphics capability. Like low-resolution graphics, you have two separate areas or pages available for use. However, this is where the similarity ends. Resolution in this mode is 280 horizontal positions by 192 vertical positions, an increase over low-resolution by 7 times on the horizontal axis and 4 times on the vertical axis. Although fewer colors are available for you to use in high-resolution graphics mode, you can plot much finer lines on the screen.

Built-in high-resolution graphics functions are only available in Applesoft. Integer BASIC has no intrinsic commands of this kind. However, no matter what language you use on the Apple II, the computer always has the capability for high-resolution graphics because the Apple II uses part of its memory to store high-resolution dots, lines, and shapes and includes built-in programs which interpret and display this memory on the TV screen. Certain Applesoft statements automatically use the built-in programs and screen memory; we will discuss them first. Later in this section you will see how to incorporate high-resolution graphics into an Integer BASIC program as well. As you read further, you will discover some high-resolution tricks achievable *only* with Integer BASIC.

### WHICH PAGE SHOULD YOU USE?

There are some difficulties in using high-resolution graphics, and they involve how much memory your Apple II has. If you have firmware Applesoft (i.e., in ROM, or the Apple II Language System), you can use high-resolution page 1 *if* your computer contains more than 16K, and you can use page 2 *if* your Apple II has 24K of memory or more. Add 12K to the minimum memory if you will use the Disk Operating System (DOS) and Applesoft high-resolution graphics at the same time.

If you use the versions of Applesoft from cassette or disk, you cannot use high-resolution page 1, as it is used to hold part of the Applesoft interpreter. If you try to use high-resolution page 1, you will corrupt or entirely lose the ability to program in Applesoft, as well as the actual BASIC program currently in memory. You *can* use high-resolution page 2 if your Apple II has at least 24K, or 36K if you want to have DOS in memory at the same time.

### Setting Aside High-Resolution Graphics Memory

Applesoft does not automatically protect memory for high-resolution graphics.

As a program becomes larger, either by adding statements or more variables, the chances of destroying graphics pages increase. The way to solve this problem is to set two memory pointers, HIMEM: and LOMEM:, to values which will protect the high-resolution graphics page or pages you use. These pointers act as boundaries which your program will not cross, thus yielding up to two areas of memory which you can use freely.

HIMEM: and LOMEM: usage differs in Applesoft and Integer BASIC. Check these differences in the BASIC compendium, located in Chapter 8. Also, Figure G-1 in Appendix G depicts memory usage pictorially. The following three paragraphs will be easier to follow if you refer to it.

If you intend to use high-resolution page 1 (remember that you must have firmware Applesoft to do this), and you only have a 16K Apple II, set HIMEM: to 8191 and leave LOMEM: alone. This keeps your Applesoft program below memory location 8191, which is a significant restriction but one that is unavoidable since you need 8K for high-resolution screen memory. If you want to use high-resolution page 1 on an Apple II with more than 16K, you may prefer to leave HIMEM: alone and set LOMEM: to 16384. This places your Applesoft program above high-resolution page 1. *CAUTION:* Do not use this latter scheme in conjunction with DOS unless you have at least 32K of memory. Otherwise, stick to the first scheme, which keeps your Applesoft program below page 1.

If you intend to use high-resolution page 2 (possible only with 24K or more), set LOMEM: to 24576, or set HIMEM: to 16383 and leave LOMEM: unchanged. By adjusting LOMEM:, you place your Applesoft program *above* high-resolution page 2. In order to use DOS at the same time, you need at least 48K of memory. If you adjust HIMEM: instead, your program resides *below* high-resolution page 2. In this latter case, if you are using a nonfirmware Applesoft interpreter (from cassette or disk), your Applesoft program will have to squeeze in above the interpreter and below high-resolution page 2, a tight fit.

If you plan to use both pages for high-resolution graphics (possible only with firmware Applesoft), set LOMEM: to 24576 or set HIMEM: to 8191. If you put your Applesoft program above the high-resolution pages by adjusting LOMEM: and leaving HIMEM: alone then you will need at least 48K in order to use DOS and still have a meaningful amount of memory for your Applesoft program.

## SETTING UP THE GRAPHICS DISPLAY

Although two pages are available for high-resolution graphics, cassette and disk Applesoft use part of high-resolution page 1 for storing the language itself (the interpreter). If you have an Apple II Plus or a standard Apple II, either with the Applesoft Firmware card or the Language System, you can use page 1 without adversely affecting BASIC. For the sake of compatibility with other Apple II computers, you may want to use high-resolution page 2 anyway.

In Applesoft, the statement:

HGR

clears and then displays high-resolution page 1, with a four-line text window at the bottom of the screen. You can have both graphics and text on the screen in this mode by using PRINT statements in Applesoft to display text in the text window. However, once the Apple II executes HGR, the screen will show only 160 of the 192 high-resolution horizontal lines available. In order to show full-screen graphics, perform a POKE –16302,0 after HGR. This will eliminate the text window and replace it with the remaining 32 lines of graphics.

To display high-resolution page 2, use the statement:

HGR2

Execution of HGR2 erases and then displays all 192 lines of high-resolution page 2, leaving no text window at the bottom of the screen. To open the text window at the bottom of the screen, perform a POKE –16301,0 after HGR2. The text window is more difficult to use with high-resolution page 2, however. The text which appears in the window is from the secondary text page. BASIC can only access this page via POKE statements (*not* via PRINT), which is quite limiting. Furthermore, the secondary text page is not protected from overwriting by BASIC as is text page 1, so you will need to set LOMEM: to 3071 or greater.

## ALTERNATIVES TO HGR AND HGR2

A principal disadvantage of using HGR and HGR2 is that executing either of these statements clears the high-resolution page selected whether you like it or not. Moreover, these statements are not available in Integer BASIC. You can use PEEK, POKE, and CALL statements to set up the graphics pages more flexibly, and you may find them useful no matter what language you use on the Apple II.

### Another Way to Set Up the Graphics Display

It is possible to go into high-resolution graphics mode without erasing the display screen. You can liken this procedure to flipping a series of switches. In theory, you are doing just that to a set of reserved memory locations called *soft switches*, which reside at memory locations –16304 through –16297 (\$C050 through \$C057). Figure E-1 in Appendix E illustrates the available switches. In order to keep these statements compatible with both Integer BASIC and Applesoft, we use the negative integer representation of these memory locations (e.g., –16304 instead of 49231).

To display high-resolution graphics page 1 without erasing its previous contents, perform the following statements:

POKE –16304,0	Sets graphics mode
POKE –16297,0	Sets high-resolution mode
POKE –16300,0	Selects high-resolution page 1 (Only necessary if switching from high-resolution page 2)

Try these statements in immediate mode as an experiment.

To display high-resolution graphics page 2 without erasing its previous contents, enter the following statements:

POKE -16304,0	Only necessary if graphics mode has not already been set
POKE -16297,0	Only necessary if high-resolution mode has not already been set
POKE -16299,0	Selects high-resolution page 2

### Normal BASIC After High-Resolution Graphics

In Integer BASIC, the TEXT and GR statements may be insufficient to completely reset the display screen. If you have been using page 2 graphics, you must explicitly reselect page 1 with a POKE -16300,0 statement. Otherwise you will see page 2 of the text/low-resolution screen memory. This can be especially confusing in text mode. The keyboard appears to be dead because everything you type goes into page 1 screen memory, while you are looking at page 2 screen memory.

The GR statement does not switch from high-resolution to low-resolution graphics in Integer BASIC. It only selects graphics mode with a four-line text window (as opposed to all-text mode). You must explicitly select low-resolution graphics with a POKE -16298,0 statement.

### Clearing the High-Resolution Pages

If you use high-resolution graphics under Applesoft, HGR or HGR2 will clear the selected page whenever the statement is executed. However, under Integer BASIC there is no single statement which performs this function. The following subroutine uses a built-in Monitor function to clear the high-resolution screen:

```

18990 REM *****
18991 REM *   CLEAR HI-RES SCREEN   *
18992 REM * (USES MONITOR'S "MOVE" *
18993 REM * SUBROUTINE AT $FE2C *
18994 REM * TO MOVE DATA QUICKLY *
18995 REM * THROUGH THE HI-RES AREA)*
18996 REM * -----*
18997 REM * SET PAGE=1 OR 2; THE *
18998 REM * ROUTINE DOES THE REST. *
18999 REM *****
19000 START=32
19010 IF PAGE=2 THEN START=64
19020 POKE 60,0
19030 POKE 61,START
19040 POKE 62,254
19050 POKE 63,START+33

```



```

19060 POKE 66,1
19070 POKE 67,START
19080 POKE -16304,0
19090 POKE -16297,0
19100 IF PAGE=1 THEN POKE -16300,0
19110 IF PAGE=2 THEN POKE -16299,0
19120 POKE START*256,0
19130 CALL -468
19140 RETURN

```

This BASIC subroutine moves zeros throughout the high-resolution graphics page you select. If the variable PAGE is set to 1, the subroutine clears page 1; if set to 2, page 2 is cleared. This subroutine only works in Integer BASIC; however, don't consider that a disadvantage. Applesoft performs this function much more efficiently with HGR or HGR2.

## HIGH-RESOLUTION COLORS

Eight color choices are available in high-resolution mode, but there are only four different colors available, plus black and white. Table 6-2 shows these colors and their corresponding numbers (used when selecting the color).

### The HCOLOR Statement

The Applesoft HCOLOR statement selects one of the eight colors available for high-resolution use. Unlike the COLOR statement in low-resolution graphics (which will allow you to specify numbers higher than those allocated to colors), HCOLOR will not accept a color number greater than 7. If you do specify an out-of-range color, your program will abruptly stop with an ?ILLEGAL QUANTITY ERROR message. HCOLOR does not change the color of any graphics already on the high-resolution screen, nor does it have any effect on low-resolution graphics.

### Setting a High-Resolution Background Color

With a slight modification, the subroutine presented earlier which clears the high-

TABLE 6-2. High-Resolution Graphics Colors

Color	Number	Color	Number
Black	0	Black	4
Green	1	Orange	5
Violet	2	Blue	6
White	3	White	7

resolution screen will fill the entire screen with any one of the high-resolution colors. The subroutine below does this. Before calling the subroutine, assign the graphics page number to variable PAGE. Also, assign the high-resolution color number to variable HCOLR. The calling program must set the soft switches for high-resolution graphics mode. This subroutine works only in Integer BASIC.

```

18990 REM *****
18991 REM * SET BACKGRND COLOR*
18992 REM *-----*
18993 REM * SET PAGE=1 OR 2; *
18994 REM * ALSO SET HCOLR TO *
18995 REM * APPLESOFT EQUIVA- *
18996 REM * LENT HI-RES COLOR.*
18997 REM *****
19000 START=32
19010 IF PAGE=2 THEN START=64
19020 POKE 60,0
19030 POKE 61,START
19040 POKE 62,253
19050 POKE 63,START+33
19060 POKE 66,2
19070 POKE 67,START
19075 Y1=85:X1=42
19080 IF HCOLR>0 AND HCOLR<>4 THEN 19090:X1=0:Y1=0
19090 IF HCOLR<>3 AND HCOLR<>7 THEN 19100:X1=127:Y1=127
19100 IF HCOLR=1 OR HCOLR=5 THEN 19120
19110 X3=X1:X1=Y1:Y1=X3
19120 IF HCOLR<4 THEN 19140
19130 Y1=Y1+128:X1=X1+128
19140 POKE START*256,X1
19150 POKE START*256+1,Y1
19160 IF PAGE=1 THEN POKE -16300,0
19170 IF PAGE=2 THEN POKE -16299,0
19180 CALL -468
19190 RETURN

```

## PLOTTING POINTS AND LINES

One powerful advantage in Applesoft high-resolution graphics is the ability to plot lines of any angle as well as individual points and horizontal or vertical lines. The HPLOT statement can be used in three ways:

```
HPLOT 12,12
```

This plots a single point on the currently selected high-resolution page at the intersection of the thirteenth row and thirteenth column, in the currently selected high-resolution plotting color.

The second use of HPLOT is:

```
HPLOT 0,0 TO 279,191
```

This statement draws a diagonal line from the upper righthand to lower lefthand corners of the screen. Using HPLOT with two sets of coordinates as shown above, you can plot from one point to another on the screen.

The third type is more sophisticated:

```
HPLOT 0,0 TO 279,0 TO 279,191 TO 0,191 TO 0,0
```

This version of HPLOT only works on firmware versions of Applesoft. Cassette-based and disk-based Applesoft do not allow this form of HPLOT. Here you can define multiple plotting statements very easily. All segments are drawn in the same color. This can be quite useful if you want to draw a many-sided shape.

### Alternatives to HPLOT

The subroutine listed below allows you to program high-resolution graphics in Integer BASIC, without having to switch pages. You may find this subroutine useful; however, it is rather slow because all calculations occur in BASIC. To use this subroutine, pass the X and Y coordinates of the point you wish to plot. Make sure that PAGE is set to the high-resolution page you want to use. It is up to you to set high-resolution, graphics, and full-screen graphics modes. This adds some flexibility. If you use this subroutine *without* first setting high-resolution mode, the subroutine will plot into high-resolution screen memory but will not change the screen display itself. Later, after the program finishes plotting in the screen memory, it can turn on high-resolution graphics with a series of POKE statements (as described earlier) and the high-resolution plotting that took place behind the scenes will suddenly be visible. This lets you program points on page 2 while displaying page 1 high-resolution graphics, and vice versa.

```
20000 REM *****
20001 REM * HI-RES INTEGER PLOT *
20002 REM * ----- *
20003 REM * SET X=COL, Y=ROW, *
20004 REM * PAGE=1 OR 2; USES *
20005 REM * VARS Y1,X1,X3 *
20006 REM *****
20007 REM
20010 Y1=PAGE*8192: REM SET BASE ADDRESS
20020 Y1=Y1+(Y/64)*40+(Y MOD 8)*1024
20030 Y1=Y1+(Y MOD 64/8)*128+X/7
20040 X1= PEEK (Y1): REM READ IN THE HI-RES BYTE
20050 X3=2 ^ (X MOD 7)
20060 REM 'OR' THE BYTE IN X1 WITH
20070 REM THE BIT VALUE IN X3.
20080 IF X1 MOD (X3 ^ 2)<X3 THEN X1=X1+X3
20090 POKE Y1,X3
21000 X1= PEEK (Y1)
21010 X3=2 ^ (X MOD 7)
21020 IF X1 MOD (X3*2)<X3 THEN X1=X1+X3
21025 X3=2 ^ (X MOD 7)
21026 GOTO 21100
21030 POKE Y1,X3
21040 RETURN
21100 POKE Y1,X3
21130 RETURN
25000 GOSUB 19000
```

### An Integer BASIC High-Resolution Example

The next program uses two of the subroutines recently introduced to plot points on the high-resolution screen. The game controls determine the coordinates to plot. By rotating the controls, you can sketch lines on the screen.

```
10 REM THIS PROGRAM USES SPECIAL
20 REM SUBROUTINES TO CLEAR AND
30 REM PLOT IN HI-RES GRAPHICS
40 REM USE CTRL-C TO END PROGRAM
89 REM SET GRAPHICS MODE
90 POKE -16304,0
99 REM SET HI-RES GRAPHICS
100 POKE -16297,0
109 REM SELECT FULL-SCREEN GRAPHICS
110 POKE -16302,0
200 PAGE=2
204 REM CLEAR HI-RES SCREEN MEMORY
205 GOSUB 19000
209 REM GET POINT COORDINATES
210 X= PDL (1)
220 Y= PDL (0)
229 REM PLOT HI-RES POINT
230 GOSUB 20010
239 REM GET MORE COORDINATES
240 GOTO 210
260 END
```

For an interesting variation of this program, try using the subroutine which fills in a solid color on the screen instead of the clear-screen subroutine.

Try improving the program by checking the game control pushbuttons with POKE statements, and clearing the screen each time a pushbutton is pushed.

## USING HIGH-RESOLUTION SHAPES

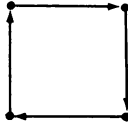
Along with coordinate plotting and drawing, the Apple II enables you to define, draw, and manipulate two-dimensional shapes in high-resolution graphics mode. This section describes how to create, design, and use a shape under Applesoft. Thorough as it may be, this section only begins to explore the creative possibilities open to you.

If you have written any high-resolution graphics programs which plot geometric figures, you probably encountered some difficulty in manipulating those figures on the screen. For instance, you may want to rotate the figure on an axis, or make it appear larger or smaller on the screen. High-resolution shapes have this manipulation feature.

### DEFINING SHAPES

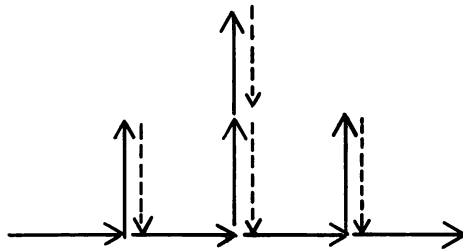
High-resolution shapes require planning. In essence, you go beyond telling the

computer to draw a line from point A to point B. When you use shapes on the Apple II, you describe the entire figure before instructing the computer to draw it. You define high-resolution shapes in a *shape table*, so called because it contains the coded characteristics of the figure to draw. The first step in defining a high-resolution shape is to draw the shape itself on paper. Take the example of drawing a square, which consists of four lines of equal length, each one at a right angle to the previous line drawn:



The shape table contains coded instructions to draw a figure; these instructions are called *plotting vectors*. Each vector describes movement up, down, left, right, or not at all, and also whether to draw on the screen or not. You can interpret each side of the square in the illustration above as a direction in which to draw: one up, one right, one down, and one left. This is the way Applesoft's shape manipulation routines look at figures.

Figures are more difficult to draw if they contain diagonal lines or curves. A triangle, although it has one side less than the square, involves much more work because it contains at least one diagonal line. The broken lines in the illustration below indicate movement without any drawing (*ghost vectors*):



Since you can only define a shape with vectors which move up, down, or sideways, some shapes, such as circles, may not be worth approximating. In some cases it may be easier to draw complicated shapes using HPlot rather than using shape tables.

## ASSEMBLING THE SHAPE TABLE

The figure you drew on paper must go through a conversion to coded plotting vectors. This section tells you how to make the conversion. The next section presents an Applesoft program that does the conversion for you. So you may skip to the next section if you are not interested in all of the inner workings of shape tables.

Vector codes range in value from 0 to 7; each byte of a shape definition (part of the shape table) can hold up to three vectors. Table 6-3 shows the possible plotting vector codes. Once the shape is reduced to a set of vectors, the vectors can be placed in memory, where certain Applesoft commands can decode them and draw the shape.

Pick a starting point on the shape. Make a list of the plotting vectors needed to construct the shape, using arrows (↑ ← ↓ →). List the vectors in order as you go around the shape (clockwise or counterclockwise, it doesn't matter). Mark any vectors to be plotted but not drawn (ghost vectors). Starting in the lower lefthand corner, our square corresponds to these vectors:

Direction	Plot
↑	Yes
→	Yes
↓	Yes
←	Yes

Now write the proper binary code next to each vector (use Table 6-3 to translate). This is what you should get:

Vector	Code
↑	100
→	101
↓	110
←	111

As shown in Table 6-4, each byte of the shape table contains three sections, each of which may contain a plotting vector. Notice that sections 1 and 2 contain three bits each, while section 3 only contains two bits.

TABLE 6-3. Plotting Vectors and their Binary Codes

Symbol	Action	Binary Code	Decimal Code
↑	Move up without plotting	000	0
→	Move right without plotting	001	1
↓	Move down without plotting	010	2
←	Move left without plotting	011	3
↑	Move up with plotting	100	4
→	Move right with plotting	101	5
↓	Move down with plotting	110	6
←	Move left with plotting	111	7

TABLE 6-4. Shape Table Byte

Bit	Section 3		Section 2			Section 1		
	7	6	5	4	3	2	1	0
M = Movement bit P = Plot/No Plot bit	M	M	P	M	M	P	M	M

By inspecting Table 6-3, you can see that some of the codes used for drawing are three-bit numbers. This is fine for sections 1 and 2 of each shape definition byte, each of which holds three bits. However section 3, because it only contains two bits, can only hold certain plotting vectors. The vectors allowed in section 3 are right, left, and down without plotting. No other plotting vectors are recognized in section 3.

Most of the time you will find that section 3 goes unused. This does not mean you can forget about using it altogether, but in most cases you can. If section 3 of a shape definition byte is set to zero, Applesoft ignores the section, moves on to the next byte of the shape definition and interprets it for drawing.

Plotting vectors equal to zero can mean two things. In section 3 of each shape definition, a zero plotting vector always means "no movement and no plotting." However, in Table 6-3, a zero vector means "move up without plotting." This ambiguity can cause problems in sections 1 and 2 of each shape definition byte, because under certain circumstances Applesoft ignores zero plotting vectors, and in others it performs upward movement without plotting. The rule here is to never end a shape definition byte with a zero plotting vector if you intend the zero vector to mean "move up without plotting." Applesoft's shape manipulation routines assume that if the most significant portion (section 3) or portions (sections 2 and 3 together) are set to 0, no drawing action takes place at all for those sections set to 0.

If all three sections of a shape definition byte are set to 0, Applesoft interprets this as an "end of shape definition" signal. In fact, you must end each shape definition with a termination byte, set to 0. Otherwise, Applesoft will draw past the end of your original shape, and will continue drawing until it encounters a 0 byte.

You can use the "move up without plotting" vector as long as a different plotting vector comes after it in the same byte. For example, section 2 can be set to 0 (which is "move up without plotting") and if section 3 is set to 01, 10 or 11 (binary), section 2 will be recognized as "move up without plotting." If section 3 is set to 0 and section 2 is set to 0, no movement occurs and Applesoft looks to section 1 of the next byte for the next valid plotting vector.

Armed with this knowledge, you can now arrange the binary-coded plotting vectors for each segment of the shape into groups of two or three. In this way,

you transpose the three-bit plotting vector codes into eight-bit bytes which can be stored in memory. Plotting vectors for a square *map* into a shape definition as shown in Table 6-5.

With the shape now mapped into binary-coded bytes, you can easily convert each byte to hexadecimal notation. Appendix J contains a binary-to-hexadecimal conversion table. Table 6-5 shows the hexadecimal encoding of the square.

The shape definition is now complete. The next step is to create a series of pointers to this shape (and others — up to 255 shapes) which Applesoft uses as a directory.

### Assembling the Shape Table Directory

The directory of a shape table is a series of bytes which describes how many shapes there are in the table, and also points to each shape definition in the table. The first byte of the directory contains the total number of shapes in the table. This number ranges from 0 to 255 (\$FF). The second byte is unused and should be set to zero.

The remaining bytes in the directory contain pointers to each shape definition you have in the table. Each pointer is two bytes long and contains the offset (absolute distance in bytes) of the shape from the *beginning* of the directory. The low-order byte of the pointer precedes the high-order byte. For example, if the offset of a shape is ten bytes, you encode the pointer in hexadecimal as 0A 00. In the case of our square, there is only the one shape to list in the directory, so the offset of shape 1 from the beginning of the directory is four bytes. Therefore, 04 00 moves into this section of the directory.

Byte		
0	01	← Number of shapes in the table
1	02	
2	04	} ← Offset of shape 1 from byte 0 (low-order byte first)
3	00	
4	25	} ← Shape definition
5	37	
6	00	← Shape ends with 00

It is good practice to leave extra bytes at the end of a shape table directory to allow room for future shape table pointers. If you have no room at the end of the directory to allow for expansion, you will have to reorganize the entire shape table in order to insert a new shape pointer. Even though you may only need a directory which holds ten shapes, you should leave unused space at the end of the directory; 20 extra bytes allow for another ten shape pointers which you can use later. When you want to add another shape to the table, place the new shape definition just after the last shape definition in the table, calculate the offset of the new



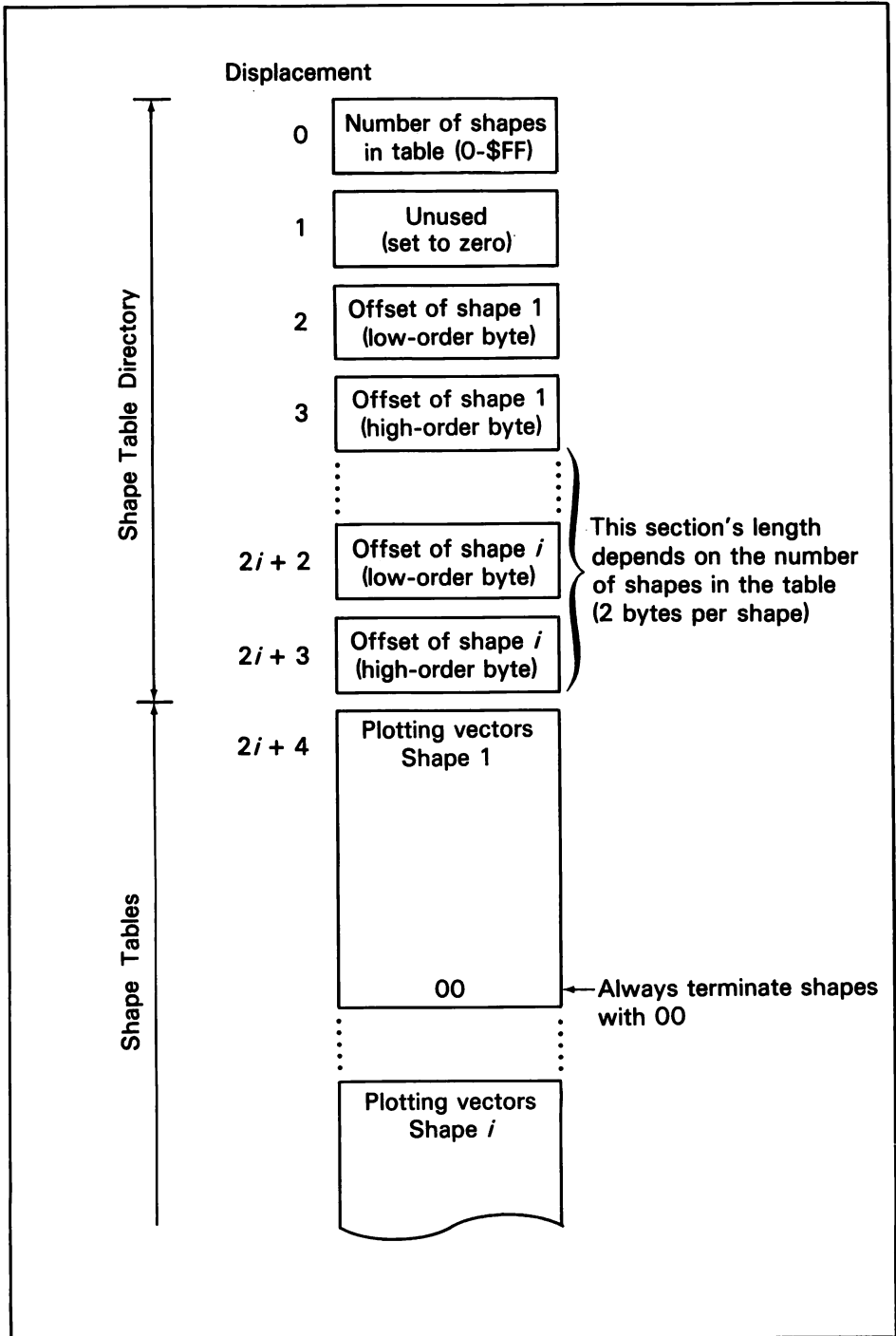


FIGURE 6-3. Shape Table Organization

TABLE 6-5. Coding a Shape Table (Square Shape)

	Vectors			Binary Codes			Hexadecimal Codes
	Sec.1	Sec.2	Sec.3	Sec.1	Sec.2	Sec.3	
Byte 0	None	↑	→	00	100	101	25
Byte 1	None	↓	←	00	110	111	37
Byte 2	None	None	None	00	000	000	00
Byte 3	—	—	—	—	—	—	—

shape from the beginning of the directory, place the new pointer immediately after the last shape pointer in the directory, and add 1 to byte 0 of the directory (which contains the number of shapes in the table).

Figure 6-3 illustrates the way shape tables and their directory are organized in memory.

### Assembling Vectors by Computer

The following program, written in Applesoft, assembles a shape definition for you. The program asks you to enter each plotting vector and whether or not to plot it. After entering the last vector, enter E for "end" and press RETURN. The program asks you to enter which vector, if any, to change. If you made any mistakes entering plotting vectors, you can correct them by entering the number of the plotting vector and then reentering the vector and whether or not to plot.

If you have no further corrections to make, enter 0 as the response to VECTOR TO CHANGE (0=END). After a few seconds, the plotting vectors display in hexadecimal notation. Here is a listing of the program and its accompanying sample run:

```

1  REM *****
2  REM * SHAPE CREATION PROGRAM *
3  REM *
4  REM *****
10 DIM S1(100),V1(100)
20 I = 0
30 PRINT "CREATE SHAPE VECTORS"
40 PRINT
41 REM ENTER PLOT ACTIONS
50 V = I: GOSUB 270
58 REM CONTINUE ENTRY UNTIL M$
59 REM EQUALS TERMINAL VALUE "E"
60 IF M$ < > "E" THEN S1(I) = M:I = I + 1: GOTO 50
70 PRINT
71 REM ALLOW CORRECTIONS
80 INPUT "VECTOR TO CHANGE (0=END):";V
90 IF V > 0 THEN V = V - 1: GOSUB 270:S1(V) = M: GOTO 80
99 REM PACK VECTORS INTO V1() ARRAY
100 FOR V = 0 TO I

```

```

110 IF B = 2 AND S1(V) > 0 AND S1(V) < 4 THEN 140
120 IF B < 2 AND (S1(V) > 0 OR S1(V) > 4) THEN 140
130 B = 0:Q = Q + 1
140 V1(Q) = V1(Q) + S1(V) * (8 ^ B)
150 B = B + 1
160 IF B > 2 THEN B = 0:Q = Q + 1
170 NEXT V
178 REM DISPLAY THE VECTORS AS
179 REM HEXADECIMAL NUMBERS
180 PRINT "BYTE","VECTOR"
190 FOR V = 0 TO Q
200 H% = V1(V) / 16
210 L% = V1(V) - H% * 16
220 IF H% > 10 THEN H% = H% + 7
230 IF L% > 10 THEN L% = L% + 7
240 PRINT V, CHR$(H% + 176); CHR$(L% + 176)
250 NEXT V
260 END
269 REM VECTOR INPUT SUBROUTINE
270 PRINT "VECTOR ";V + 1;" : ";
280 INPUT "MOVE: U/D/L/R? ";M$
290 M = 0
300 IF M$ = "R" THEN M = 1
310 IF M$ = "D" THEN M = 2
320 IF M$ = "L" THEN M = 3
330 IF M$ = "E" THEN RETURN
340 INPUT "PLOT (Y=YES,N=NO)? ";P$
350 IF P$ = "Y" THEN M = M + 4: RETURN
360 IF P$ = "N" THEN RETURN
370 GOTO 340

JRUN
CREATE SHAPE VECTORS
VECTOR 1:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? N
VECTOR 2:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? N
VECTOR 3:MOVE: U/D/L/R? L
PLOT (Y=YES,N=NO)? Y
VECTOR 4:MOVE: U/D/L/R? L
PLOT (Y=YES,N=NO)? Y
VECTOR 5:MOVE: U/D/L/R? U
PLOT (Y=YES,N=NO)? N
VECTOR 6:MOVE: U/D/L/R? U
PLOT (Y=YES,N=NO)? Y
VECTOR 7:MOVE: U/D/L/R? U
PLOT (Y=YES,N=NO)? Y
VECTOR 8:MOVE: U/D/L/R? U
PLOT (Y=YES,N=NO)? Y
VECTOR 9:MOVE: U/D/L/R? R
PLOT (Y=YES,N=NO)? N
VECTOR 10:MOVE: U/D/L/R? R
PLOT (Y=YES,N=NO)? Y
VECTOR 11:MOVE: U/D/L/R? R
PLOT (Y=YES,N=NO)? Y
VECTOR 12:MOVE: U/D/L/R? R

```

```

PLOT (Y=YES,N=NO)? Y
VECTOR 13:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? N
VECTOR 14:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? Y
VECTOR 15:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? Y
VECTOR 16:MOVE: U/D/L/R? D
PLOT (Y=YES,N=NO)? Y
VECTOR 17:MOVE: U/D/L/R? L
PLOT (Y=YES,N=NO)? N
VECTOR 18:MOVE: U/D/L/R? L
PLOT (Y=YES,N=NO)? Y
VECTOR 19:MOVE: U/D/L/R? E

VECTOR TO CHANGE (0=END):0
BYTE          VECTOR
0             12
1             3F
2             20
3             64
4             2D
5             15
6             36
7             1E
8             07
9             00

J

```

## ENTERING THE SHAPE TABLE

Before you can display any shapes which you code, you have to enter them into the computer's memory. In order to do this, you will need to determine what area of memory the shape table will reside in. The easiest way to allow for this space is to reset the HIMEM: pointer to a value just below the starting address of DOS, or just before the high-resolution graphics page you want to use. You must reset HIMEM: *before* you execute any Applesoft statements that use strings. If you use disk-based Applesoft, you will need at least 36K of memory (although 48K is better) to allow for the Applesoft interpreter and DOS. Addresses 115 and 116 (\$73 and \$74) contain the latest HIMEM: setting for Applesoft, stored low-order byte first. To calculate the new HIMEM: value which allows for the shape table, use the following statement:

```
PRINT PEEK(116) * 256 + PEEK(115) - X
```

This statement computes the HIMEM: value you should set, based on the parameter *X* which serves as the length of the shape table, including directory. Using this statement, replace *X* with the length of the shape table. Set HIMEM: to the computed value before entering the shape table into memory. This will protect the shape table from being overwritten by Applesoft.

As an alternative, you can place your shape table in memory between locations 768 and 975 (\$300 and \$3CF) inclusive. Be sure the shape table does not conflict with any machine language subroutines that you or your Applesoft program might put there.

You can use POKE statements to place the shape table in memory. For example, the following series of POKE statements puts the shape table for our square in memory starting at location 768:

```

1POKE 768,01
1POKE 769,00
1POKE 770,04
1POKE 771,00
1POKE 772,37
1POKE 773,55
1POKE 774,00

```

You can also enter a shape table from the Monitor. Use the statement CALL -151 to switch over to the Monitor. Then enter the *hexadecimal* memory address where the shape table will start, follow that with a colon, then enter the first byte of the shape table directory, enter a blank space, then enter the next directory byte followed by another space, and so on. Press RETURN. Now enter another colon followed by each hexadecimal byte of the first shape table (separate the bytes with spaces), and press RETURN. Repeat this last step for each shape in the directory. You can review your work by typing the hexadecimal starting memory address, a period, and the hexadecimal ending address of the shape table, then pressing RETURN. For more information on how to use the Monitor, see Chapter 7. This is how you would enter the shape table for our square:

```

1CALL -151

*6000:01 00 04 00           Monitor prompt
*:2C 3E 00                  Enter the shape table directory
*6000.6006                  Enter shape 1
6000- 01 00 04 00 2C 3E 00  Check entries by displaying
*                             memory

```

The shape table is now in memory. The first entry starts with the beginning address of the shape table (in this case, \$6000). The colon (:) tells the Monitor to place the series of hexadecimal digits into memory. Immediately after the colon comes the shape table directory: 01 (the number of shapes in the table), 00 (the second byte is unused), 04 and 00 (the offset of shape 1 from the beginning of

the directory — the low-order byte comes first). The next line starts with a colon; no starting address is necessary if you set it in a previous entry. The Monitor will place the next series of hexadecimal digits immediately following the first series entered.

The last line tells the Monitor to display memory addresses \$6000 through \$6006. The format for this command is: starting address, followed by a period (which tells the Monitor to display memory), followed by the last address to display. Carefully check these entries for accuracy every time you use the Monitor to enter shape tables.

### Saving the Shape Table on Tape or Disk

If you have invested a lot of time putting together shape tables, it certainly would be a good idea to save your work on magnetic media rather than lose it when you turn the Apple II off. You can use a cassette or disk to save a shape table; disk is preferable by far because it records and retrieves data much more quickly than tape.

Applesoft has a command called SHLOAD which works exclusively with tape. Before you can save the shape table on tape, you must calculate the length (in hexadecimal) of the table. Continuing with the square as our example, look at the number of bytes taken up by the shape table; the total length is 7 bytes. Using the Monitor, enter the length of the table as a two-byte hexadecimal number, starting at address 0:

```
*00:07 00
```

As always with two-byte quantities entered in the Monitor, the low-order byte comes first. Rewind the tape to the beginning. Then start it in RECORD mode. You will use the Monitor to record the shape table length and the table itself. This is done in two write operations, both of which you can enter on the same line:

```
*0.1W 6000.6006W
```

The first write command places the two-byte shape table length you entered (at memory locations 0 and 1 above) on the tape. The second write operation places the seven bytes of the shape table, from address \$6000 to \$6006, on the tape. The two write operations take just over 20 seconds. The speaker beeps twice during the recording process. After the second beep, stop the recorder. The shape table is now on tape for you to use later. Use your actual shape table length, starting location, and ending location in place of the examples.

To use the disk to save shape tables, get into Applesoft under DOS (from the Monitor, enter 3DOG). The command you use to save shapes is BSAVE. You need to know the starting address and length of the shape table you entered. To save the shape table starting at address \$6000, the DOS command:

```
BSAVE SQUARE, A$6000, L7
```

creates the disk file SQUARE with filetype B for binary. The shape table is now saved on disk, ready for you to use at a later time. Use your actual shape table name, starting location, and length in place of the examples.

### Loading Shape Tables from Tape or Disk

If you saved a shape table on cassette, Applesoft will read the tape back into memory. First, rewind the tape to the beginning. Now type in:

SHLOAD

Press the PLAY button on your tape recorder. The Apple II speaker beeps twice and then returns control of the computer to you. If some problem exists on the tape, you may see the cryptic message ERR on the display screen. If so, try reading the tape back in again, or, if the error persists, check the volume setting on the cassette recorder, as described in Chapter 2. SHLOAD automatically sets HIMEM: to the current HIMEM: value minus the length of the shape table in bytes. Make sure that the HIMEM: setting you have is correct.

If you recorded the shape table on diskette, set HIMEM: before trying to read the table into memory. The following command reads the shape table in from disk:

BLOAD SQUARE

DOS remembers what address you saved from (\$6000 in our example) and will put the shape table there automatically. If you want to read the table in starting at a different address, say \$3000, enter:

BLOAD SQUARE, A\$3000

Here again, use the actual name and optional starting address in place of the example.

After you BLOAD the shape table, you need to place the address of the shape table in addresses 232 and 233 (\$E8 and \$E9). Applesoft uses these locations to point to the shape table in memory (SHLOAD sets this address pointer automatically). Enter the Monitor and set the address (\$6000 as an example):

E8:00 06

Alternatively, you can use POKE statements to place the starting address of your shape table in locations 232 and 233. Remember that with POKE you must use *decimal* values.

You are now ready to use the shape table in an Applesoft program.

### SHAPE DRAWING COMMANDS

Applesoft has four shape manipulation statements which draw, erase and change the orientation of shapes:

DRAW, which displays the shape on the Apple II display screen.

XDRAW, which can erase shapes drawn.

ROT, which rotates the shape on the X and Y axes.

SCALE, which alters the size of the shape drawn.

The shape manipulation commands use the currently selected high-resolution graphics page (using HGR or HGR2) and color (using HCOLOR).

### The SCALE Command

As a programmed or immediate mode statement, you should always set SCALE before drawing a shape for the first time in a program.

SCALE=1

sets the scaling to draw one point for each plotting vector. If SCALE=5, the Apple II will draw 5 positions for each single plotting vector. You can set SCALE as high as 255 (255 points plotted for each vector). The maximum scale setting is SCALE=0, which plots 256 points for each single plotting vector.

### The DRAW Command

DRAW plots the shape (numbered from 1 to 255) from the shape table, in the last color chosen, at the scale and rotation value last set. This statement:

DRAW 1 AT 140,96

plots the first shape definition in the shape table starting at the 141st column and 97th row of the high-resolution display. Drawing originates at the column and row coordinates given in the statement. A second option of the DRAW statement uses an implied starting location.

DRAW 11

This statement draws the eleventh shape in the table at the point last plotted by the most recent HLOT or DRAW statement executed. In case the coordinates were not set before, Applesoft defaults to zero coordinate values.

**IMPORTANT:** Applesoft assumes that the shape table is properly located in memory. Before you execute a DRAW statement, make sure the shape table is in memory and that addresses 232 and 233 (\$E8 and \$E9) point to the beginning of the shape table. If you specify a shape number greater than the number of shapes actually in the table, or if the DRAW statement uses row or column coordinates which are not valid, drawing does not occur; instead, the error message ?ILLEGAL QUANTITY ERROR displays on the screen.

### The XDRAW Command

This statement allows you to erase a shape without erasing any high-resolution background graphics. Here is an example:

XDRAW 8 AT 90,96

This statement is syntactically identical to DRAW; the plotting coordinates can be explicit as shown above or implicit as shown in the last DRAW statement example. XDRAW checks the color of the plotting coordinates and draws a shape in the *complement* of the color found. In the example above, XDRAW occurs at the 91st row and 97th column on the screen. Table 6-6 lists the complements of high-resolution colors.

If the coordinates, rotation, and scale are the same as those of a shape already on the screen, XDRAW erases the shape, leaving all surrounding graphics intact.



TABLE 6-6. Draw Colors

If Color Is	XDRAW Color Is
Black	White
White	Black
Violet	Green
Orange	Blue
Green	Violet
Blue	Orange

### The ROT Command

ROT rotates the shape about the center of the screen (on the X and Y axes). The statement:

```
ROT=16
```

sets the angle of shape rotation to 90 degrees clockwise. The values for ROT range from 0 to 255, although there are only 64 possible rotation settings, from 0 to 63. Figure 6-4 shows the changes in axis orientation based on ROT values.

When SCALE is set to 1, ROT only rotates shapes in 90 degree increments, which means that only four meaningful rotations can be had: 0=0 degrees, 16=90 degrees, 32=180 degrees, and 48=270 degrees. Applesoft rounds the rotation value you set to the next lowest ROT increment. All 64 rotational positions are available if SCALE is set to 5 or greater.

### Using Shapes in a Program

The following program listing uses shapes and should serve as a good example of how to use them in a program:

```

1  LOMEM: 24600
20  HGR2
50  REM SET SHAPE TABLE START ADDRESS
60  POKE 232,0
70  POKE 233,96
75  REM USE ALL COLORS IN TURN
76  FOR H = 1 TO 7
79  REM INCREMENT ROTATION FACTOR
80  FOR I = 1 TO 80
82  HCOLOR= H
90  ROT= I
92  IF I < 50 THEN SCALE= I
105 DRAW 1 AT 140,96
106 ROT= I + 32
110 DRAW 1 AT 140,96
130 NEXT I
140 NEXT H
150 GOTO 76

```

Working with high-resolution shapes is especially useful in game programs.

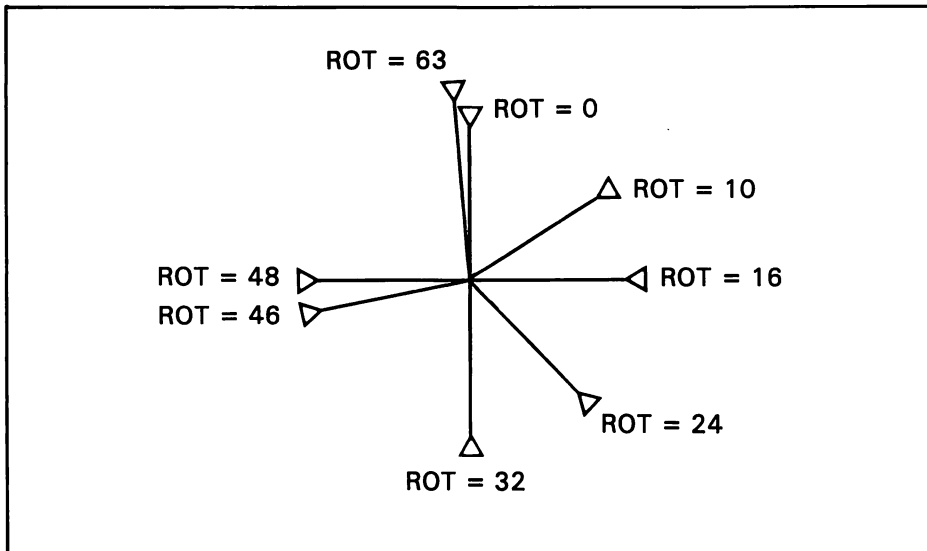


FIGURE 6-4. Shape Rotation

You can build a library of shapes without having to heavily document them as you would H PLOT statements. Shapes cut down on memory consumption. Fewer calculations are necessary to manipulate the size and orientation of the high-resolution objects you create, and the calculations themselves are simple when compared to a series of H PLOT statements designed to display a shape. If you had some difficulty getting through this section on high-resolution shapes, it is probably because you didn't try constructing shapes of your own. This is fairly complex material because so many steps are necessary to make even a single shape appear on the screen. If you use the program presented earlier which automatically encodes shapes, you will save some drudgery.

## APPLE II SOUND

After the previous sections in this chapter it may be a relief as well as an annoyance to discuss programming the Apple II onboard speaker. It is a relief because driving the speaker is really very simple. It can be an annoyance because you cannot control what the speaker does with BASIC statements. You have to define every sound the Apple II speaker makes in thorough detail. In essence, all you can do in programming the speaker is to make it emit a single click. The trick to making sounds with this speaker is to vary the frequency of these clicks, thus creating sounds of differing pitch. This section explains how to operate the speaker, and includes a program which creates a series of sounds.

## OPERATING THE SPEAKER

The Apple II uses memory location -16336, also known as 49200 and \$C030, as a toggle switch, much the same as the graphics switches discussed earlier in this chapter. Any time you access this location, a click emanates from the Apple II speaker. In BASIC, you can operate the speaker by using the statement:

```

      A=PEEK(49200)
or    A=PEEK(-16336)

```

Using a BASIC program to drive the speaker (shown in the short example below) is easy, but the speaker will only generate lower frequencies. This is because BASIC is comparatively slow. In Integer BASIC, the highest frequency possible is about 256 Hz (cycles per second), and in Applesoft, the highest frequency is around 72 Hz. The only way to generate sounds of any higher frequency is to use a machine language subroutine to drive the speaker.

```

      9 REM  CLICK THE SPEAKER
     10 A= PEEK (-16336)
     20 GOTO 10

```

### The Machine Language Sound Subroutine

Applesoft and Integer BASIC allow room in memory between locations 768 and 975 (\$300 and \$3CF) for machine language subroutines and shape tables without having to change LOMEM: settings. (DOS uses this area, eliminating all previous contents, when you boot a DOS master diskette.)

If you have not covered assembly language and machine language yet, you can still incorporate the following subroutine into Integer BASIC or Applesoft programs by keying in a series of hexadecimal numbers. To place the subroutine into memory, get into the Monitor by entering the command CALL -151 or by pressing RESET (*Important*: if you have DOS, make sure you boot it before entering the Monitor). Next, enter the machine language subroutine as shown below:

```
>CALL -151
```

```
*F666G
```

```
!302:LDY 301
```

```
0302- AC 01 03 LDY $0301
! LDX 301
```

Computer displays shaded  
lines over your entries

```
0305- AE 01 03 LDX $0301
! LDA #4
```

```
0308- A9 04 LDA #$04
! JSR FCA8
```

```
030A- 20 A8 FC JSR $FCA8
! LDA C030
```

```

030D- AD 30 C0 LDA $C030
! INX

0310- E8 INX
! BNE 310

0311- D0 FD BNE $0310
! DEY

0313- 88 DEY
! BNE 305

0314- D0 EF BNE $0305
! DEC 300

0316- CE 00 03 DEC $0300
! BNE 302

0319- D0 E7 BNE $0302
! RTS

031B- 60 RTS

```

This subroutine uses addresses \$300 and \$301 for data storage; addresses \$302 through \$31B hold the routine itself. Once you have entered the subroutine, check it for correctness as follows:

!\$FF69G

\*302L

```

0302- AC 01 03 LDY $0301
0305- AE 01 03 LDX $0301
0308- A9 04 LDA #$04
030A- 20 A8 FC JSR $FCA8
030D- AD 30 C0 LDA $C030
0310- E8 INX
0311- D0 FD BNE $0310
0313- 88 DEY
0314- D0 EF BNE $0305
0316- CE 00 03 DEC $0300
0319- D0 E7 BNE $0302
031B- 60 RTS
031C- 20 20 70 JSR $7020
031F- 08 PHP
0320- 18 CLC
0321- D8 CLD
0322- 88 DEY
0323- 08 PHP
0324- A0 A0 LDY #$A0
0326- 10 38 BPL $0360

```

\* ← CTRL-B into BASIC here

If you have DOS on your Apple II, you can save this subroutine on a disk with the BASIC command BSAVE:

BSAVE SOUND,A\$302,L26

This creates a binary file entitled **SOUND**; you can load this subroutine at a later time. If you have a cassette, record the routine by entering the following Monitor command:

```
*302.329W
```

The subroutine is saved on tape for reading in at a future time.

### The BASIC Interface

You will need a BASIC subroutine which talks to the machine language subroutine. Enter the following listing in Applesoft or Integer BASIC:

```
3200 REM  SPEAKER DRIVER
3210 POKE 768,D
3220 POKE 769,F
3230 CALL 770
3240 RETURN
```

With this subroutine working in conjunction with the machine language routine, a BASIC program can generate sounds by setting two variables: F (for frequency) can range from 1 to 255, with 255 as the highest pitch, and D (duration), also ranging in value from 1 to 255, with 255 the longest duration possible. As an example, enter the following BASIC program after entering the subroutine listed above:

```
10 FOR I = 1 TO 254
20 F = 1
30 D = I
40 GOSUB 3200
50 NEXT I
60 END
```

When you run the program, listen to the duration of each note. On the low-frequency and high-frequency ends of the scale, sounds are shorter than those in mid-frequency. This problem is inevitable because the machine language subroutine uses instructions rather than elapsed time to change the frequency of each note. Since there is no real-time clock on the Apple II, this approach is necessary. By setting longer duration values for very high and very low notes, you can compensate for this inequality.

### A More Elaborate Sound Program

The program listed below uses the sound generation subroutines detailed earlier to create a series of sounds which you can listen back to, change, and finally print out for use in other BASIC programs. When you run the program, you will see the prompt (E)NTER, (L)ISTEN, (P)RINT? The first action is to enter some tones; enter E and press RETURN.

Now the prompt TONE 0: FREQUENCY, DURATION? appears. Enter two numbers separated by a comma. The first is the frequency and the second is the duration. Both numbers must be between 1 and 255. When you press RETURN after

entering the frequency and duration of the tone, the Apple II speaker emits the tone. This process repeats for a series of tones (up to 100). After the last tone you enter, input a 0 tone and 0 duration to complete entry.

After you enter all the tones, the prompt WHICH NOTE TO CHANGE? appears. If you want to change any tones entered, input the number of the tone you want to reenter; otherwise, enter 0.

The prompt (E)NTER, (L)ISTEN, (P)RINT? appears again when you finish making changes; now enter L and press RETURN. The Apple II now repeats the entire series of tones you entered. When the last note plays, the prompt message appears again. Enter P to display or print the frequency and duration values of each tone.

The program listing below can easily be modified to allow for saving the tone values on cassette or disk. Then other programs can retrieve the tone values and, using the sound generation subroutines, can play music or just emit random noises.

```

10 REM SOUND GENERATOR PROGRAM
19 REM ARRAY REMEBERS ENTERED TONES
20 DIM A(100,2)
29 REM CLEAR DISPLAY
30 CALL - 936
40 INPUT "(E)NTER, (L)ISTEN, (P)RINT?";A$
50 IF A$ = "L" THEN 1000
60 IF A$ = "P" THEN 1200
80 IF A$ < > "E" THEN 30
81 REM ENTER EACH TONE
90 PRINT
100 I = 0
105 M = I
110 GOSUB 3000
119 REM END OF TONE ENTRY?
120 IF F = 0 AND D = 0 THEN I = I - 1: GOTO 200
129 REM NO--REMEMBER TONE
130 A(I,1) = F:A(I,2) = D
140 I = I + 1
150 GOTO 105
200 REM CHANGE ANY ITEMS HERE
205 PRINT "WHICH NOTE TO CHANGE (0-";I;")";
206 INPUT E
208 IF E = 0 THEN 30
210 IF E < 1 OR E > I THEN 210
220 M = E: GOSUB 3000
230 A(E,1) = F:A(E,2) = D: GOTO 205
1000 REM LISTEN TO THE NOTES SO FAR
1010 FOR K = 0 TO I
1020 F = A(K,1):D = A(K,2): GOSUB 3200
1030 NEXT K
1040 GOTO 30
1200 REM PRINT OUT THE NOTES
1210 PRINT "NOTE#","FREQ","DURATION"
1220 FOR K = 0 TO I
1230 PRINT K,A(K,1),A(K,2)
1240 NEXT K

```

```
1250 PRINT
1260 GOTO 30
3000 PRINT "TONE ";M;
3010 INPUT " ENTER FREQUENCY, DURATION";F,D
3015 IF F = 0 AND D = 0 THEN RETURN
3020 IF (F < 0 OR F > 255) OR (D < 1 OR D > 255) THEN 3010
3030 GOSUB 3200
3040 RETURN
3200 REM SPEAKER DRIVER
3210 POKE 768,D
3220 POKE 769,F
3230 CALL 770
3240 RETURN
```





# 7

## Machine Language Monitor

Residing permanently in the Apple II read-only memory (ROM) is a control program called the Monitor. This chapter describes the Monitor's features and uses and shows how you can use it in conjunction with BASIC programs you write. The Monitor is written in machine language; it serves as a link between BASIC (and other languages which the Apple II computer supports) and the various low-level functions which the machine performs, such as printing a character, plotting a line, and so forth.

You can also use the Monitor via keyboard commands. Some reasons for doing this would be to create graphics shape tables (described in Chapter 6), to examine memory to isolate hardware problems, or to program in assembly language. Most of the time you will find no pressing need to use it, but the Monitor has some functions which you may find handy at some time.

After describing the Monitor and its facilities, this chapter explains how to use the Mini-Assembler. This chapter does not teach you assembly language programming; Appendix K lists several books which do. You will learn how to integrate your assembly language program with a BASIC program, and how to use the Mini-Assembler to write, test, and debug it.

### ACCESSING THE MONITOR

There are two versions of the Monitor: the standard version and the Autostart version. If your Apple II has a standard Monitor, simply turning on the computer

will place you in the Monitor. In this situation you will see a screen full of random characters, with an asterisk (\*) at the bottom lefthand corner of the screen and a blinking cursor directly to the right of the asterisk. The asterisk is the Monitor prompt character.

If your Apple II has an Autostart Monitor (either as an add-on feature or as standard equipment in an Apple II Plus), you will have to use either Integer BASIC or Applesoft to access the monitor. When you see a BASIC prompt (either > for Integer BASIC, or ] for Applesoft), type in the following command:

```
CALL -151
```

This statement is actually a subroutine call to hexadecimal address FF69, but BASIC will not recognize hexadecimal numbers, so you must use the decimal equivalent of FF69 in this command. Once you key this statement in, the asterisk prompt, followed by the cursor, appears. At this point you are in the Monitor.

## LEAVING THE MONITOR

There are a few ways in which you can exit the Monitor and return to BASIC; they largely depend on what you want to do when you leave it. To preserve program statements and variables used in the BASIC program, exit the Monitor by pressing CTRL-C and then pressing RETURN. After pressing RETURN, the prompt for BASIC returns. At this point, you can print the values of variables and list the program (if one happens to be in memory at the time).

To illustrate, let's say that you put a value in a given memory location via the POKE statement and you want to verify the POKE. Of course, you could use PEEK to do the same thing, but the Monitor allows you far more access to the Apple II memory than PEEK and POKE ever will. The Integer BASIC example below shows how using CTRL-C preserves the BASIC program and variables when leaving the Monitor:

```
>10 A=123
>19 REM MOVE CURSOR TO 13TH COLUMN
>20 POKE 36,12
>30 PRINT A
>40 END
>RUN
123

>CALL -151
* ← Press CTRL-C, then RETURN

>PRINT A
123

>LIST
10 A=123
19 REM MOVE CURSOR TO 13TH COLUMN
20 POKE 36,12
30 PRINT A
40 END

>
```

If you want to leave the Monitor and clear out the current BASIC program and variables stored in memory, press `CTRL-B` and `RETURN`. This entry will return you to BASIC, but any program statements or variables which may have been in memory will be erased. Try the example above, substituting `CTRL-B` for `CTRL-C`. After returning to BASIC from the Monitor, you will find that no value exists for `A$`, and that you cannot list the program.

`CTRL-C` works fine with Integer BASIC and *firmware* Applesoft. It does not work with cassette-based or disk-based Applesoft.

With disk-based or cassette-based Applesoft you may use `CTRL-B` to exit the Monitor. In this case you will return to Integer BASIC, and all your Applesoft program and variables will be erased. There is a way to return to Applesoft with your program and variables intact.

To return to disk-based Applesoft from the Monitor, type this command:

`*3DOG`

To return to cassette-based Applesoft from the Monitor, type this command:

`*OG`

`3DOG` and `OG` are two instances of the Monitor's branch instruction (analogous to the `GOTO` instruction in BASIC). More on this later.

**IMPORTANT:** Use the command `OG` *only* with cassette-based Applesoft.

## FUNCTIONS OF THE MONITOR

The Monitor performs a limited number of tasks, but each one is fairly powerful considering how small the Monitor program actually is. You can examine memory locations or the microprocessor registers, dump the contents of memory to the screen or to another output device such as a printer, and change memory or registers. Other functions include moving blocks of data from one address to another and comparing blocks of memory to each other. Still more miscellaneous functions appear in this chapter.

### EXAMINING MEMORY

There are three methods in the Monitor of looking at the contents of memory locations: *single-address*, *word*, and *block* modes. A single address refers to a one-byte memory location. A word is an eight-byte segment of memory, beginning with an address divisible by eight. A block is a convenient means of looking at a range of addresses, starting at one address and ending at another address higher in memory.

#### Examining Single Addresses

When you see the Monitor prompt and you want to look at a single memory loca-

tion, just type in the hexadecimal address of the location you want to see followed by the RETURN key, as in this example:

```
*FF69
```

The Monitor responds by displaying the contents of that location:

```
FF69- A9
*
```

When you enter an address, the Monitor retains it for future use as a pointer. Therefore, if you enter FF69 (or any other hexadecimal address), the Monitor will remember that location, using it as a reference for further memory examination until you change the address. To change the pointer which the Monitor uses, simply key in a new address. For example, type:

```
*300F
```

and the Monitor resets the pointer in addition to displaying the contents of address 300F.

### Examining Words of Memory

Let's say you examined the single address FF69, as in the example above, and you want to look at the next higher locations in memory. The monitor will continue examining memory if you press RETURN, as shown below:

```
*FF69
```

```
FF69- A9
* ← Press RETURN
  AA 85 33 20 67 FD
*
```

```
FF70- 20 C7 FF 20 A7 FF 84 34
* ← Press RETURN again
```

The first time you press the RETURN key in this example, six bytes of memory display on the screen. These are the contents of locations FF6A through FF6F. The second time you press RETURN an entire eight-byte word displays, but first the starting address of the word is shown (FF70) to remind you what the starting byte of this word is. All words start at memory locations which are divisible by eight, which is why an address did not print when you pressed RETURN the first time. The first RETURN merely finished off the block you had started earlier with the single-address examination.

### Examining Blocks of Memory

You can examine a large block of memory (usually more than eight bytes) in block mode. Enter the starting hexadecimal address, followed by a period, and finally the ending hexadecimal address. For example:

```
*F800.F83F
```

The Monitor responds with the contents of the requested memory locations:

```
F800- 4A 08 20 47 F8 28 A9 0F
F808- 90 02 69 E0 85 2E B1 26
F810- 45 30 25 2E 51 26 91 26
F818- 60 20 00 F8 C4 2C B0 11
F820- C8 20 0E F8 90 F6 69 01
F828- 48 20 00 F8 68 C5 2D 90
F830- F5 60 A0 2F D0 02 A0 27
F838- 84 2D A0 27 A9 00 85 30
*
```

The starting address must be greater than or equal to the ending address in order for you to see more than one address. If the ending address is lower than the starting address, only the contents of the starting address displays.

You can specify a range of addresses which exceeds the size of the Apple II screen. In this case, data scrolls off the top of the screen to make room for more at the bottom. You cannot cancel this display without pressing **RESET**. If your Apple II contains the Autostart Monitor, you can temporarily halt the display by pressing **CTRL-S**. Using **CTRL-S** will stop output to the screen, but not to other output devices, such as a printer, giving you a chance to view the screen at your leisure. Press the space bar to restart the display.

The block method of examination is typically called a *dump*. Once it finishes, the pointer which maintains the next location to examine updates itself so it is pointing to the byte after the ending address of the block examine.

A shortened form of this command uses the pointer as the starting address of the block. You just enter a period followed by the ending address of the block. For instance, if you just looked at addresses F800 through F83F, as shown in the example above, you can continue examining a block beginning with F840 and ending with F880 by entering:

```
*.F880
```

which would result in the following output:

```
F840- 20 28 F8 88 10 F6 60 48
F848- 4A 29 03 09 04 85 27 68
F850- 29 18 90 02 69 7F 85 26
F858- 0A 0A 05 26 85 26 60 A5
F860- 30 18 69 03 29 0F 85 30
F868- 0A 0A 0A 0A 05 30 85 30
F870- 60 4A 08 20 47 F8 B1 26
F878- 28 90 04 4A 4A 4A 4A 29
F880- 0F
*
```

## EXAMINING THE MICROPROCESSOR REGISTERS

At some point you may want to inspect the registers in the microcomputer itself. This is done by typing **CTRL-E**, followed by the **RETURN** key. The results look similar to this:

```
A=C0 X=B1 Y=C3 P=B5 S=F0
```

The values displayed are those stored in the Accumulator (A), Index Register X (X), Index Register Y (Y), the Program Counter (P), and the Stack Pointer (S). The values directly to the right of each equal sign are the latest values of the registers. However, they are not affected by operating the Monitor. In other words, the register contents are saved by the Monitor and remain unchanged until you either execute your own assembly language program or return to BASIC.

## ALTERING MEMORY

Altering memory is more involved than examining it. You have to specify what address to alter as well as supply the new data which goes into that address. You can alter memory by single address (one byte at a time), or you can modify a series of consecutive locations in memory, entering new data for many addresses on one command line.

### Altering Single Addresses

The first step in changing a single address is to set the Monitor's address pointer, which is the same pointer used in examining memory. Because both Monitor commands use the same pointer, you set the address to alter the same way you set the address to examine. Type in the hexadecimal address to alter, and then press RETURN. For example:

```
*1200
```

sets the address pointer to 1200 hexadecimal.

The Monitor responds with the contents of the memory location:

```
1200- 73
```

Notice that this entry produces the same result as a single-address examine command. The Monitor's response shows where the address pointer is set (1200), and also displays the current contents of the address (in this case, 73).

The next step is to alter this address to a new value. To change memory at this address, first type a colon (:) followed by the two-digit hexadecimal number you want to place at the address you just set. For example:

```
*:5F
```

The colon indicates a memory alteration command to the Monitor. The 5F indicates the new data to place at address 1200. You can alter memory using one command line instead of two, as shown below:

```
*1200:5F
```

This command line has the same effect on changing address 1200 as the two separate lines shown above. The address pointer updates to 1200, and the Monitor places 5F in that address. The address pointer moves to the next highest memory location. So if you want to change address 1201 to 7F, for example, type:

```
*:7F
```

and the Monitor automatically updates this address to hold 7F. Again, the address

pointer increments by 1 and you can alter address 1202 to some different value by continuing the process of entering data without explicitly entering the address.

### Altering More Memory

The Monitor lets you change more than one memory location at a time, provided that the addresses you change follow each other consecutively in memory (for example, addresses 1200 to 1207 inclusive). This command starts the same way as the single-address method of altering memory. You first set the address pointer if necessary. Next, use the colon to indicate to the Monitor that this is a memory alteration command. Finally, enter the data you want at each consecutive location, separating each hexadecimal number with a space.

For example, to place the quantities 00 through 07 in addresses 1200 through 1207, enter:

```
*1200:00 01 02 03 04 05 06 07
```

You can actually alter many more addresses than eight. If you set the pointer at the beginning of the command, you could enter as many as 83 values on one command line. If you do not set the pointer, you can enter up to 84 values. In either case, the command line would wrap around to encompass several display lines. This is impractical because the wraparound command line makes checking entries extremely difficult. Moreover, the only way to correct errors is to back-space to them, retyping the balance of the line after making the correction. But if you are inclined to enter so much data on one command line, the Monitor will allow you to do so.

### Checking Memory Alterations

It is good practice to check memory alterations if you want the final product (whether a graphics shape table or a series of machine instructions) to be totally accurate. To do this you will have to use one of the three possible memory examine commands discussed earlier in this chapter. To begin checking the alterations you made, you once again have to reset the address pointer to where you first made alterations.

Assuming you placed 00 through 07 at addresses 1200 through 1207, as shown in the example above, reset the pointer to 1200:

```
*1200
```

The Monitor responds with:

```
1200- 00
*
```

By pressing RETURN you can see the remaining seven addresses which you changed:

```
* ← Press RETURN
 01 02 03 04 05 06 07
*
```

If you altered more than a few locations, keep pressing RETURN until you see the

last alterations you made.

You can also use the block examine mode to check this area:

\*1200.1207

The Monitor responds with:

1200- 00 01 02 03 04 05 06 07  
\*

### Correcting Mistakes

If there are any mistakes in the addresses you altered, you can correct them individually without having to reenter all the data correctly. The simplest way to do this is to note the address of the incorrect data and enter a single-address alteration for it.

For instance, if you made a mistake entering numbers 00 through 07 in the previous example, and the error appears at address 1204, enter:

\*1204:04

to correct the error at this address. The next step is to look back at address 1204 and make sure you got it right this time. Then finish checking over any other alterations to ensure that all of the data you entered is correct.

### ALTERING THE MICROPROCESSOR REGISTERS

The process of altering the microprocessor registers is slightly different from altering memory, since the registers actually have no addresses. To alter the contents of the registers, you first have to examine them using the CTRL-E command. Immediately following a register examination command, you can change the contents of the registers by typing in a colon (signifying an alter operation to the Monitor), followed by one to five hexadecimal numbers. Separate the numbers with spaces.

The first hexadecimal number will be the new value of the Accumulator, the second number will be the new value of Index Register X, the third number becomes the value of Index Register Y, the fourth number becomes the value of the Program Counter, and the fifth number is the new value of the Stack Pointer.

You must enter values for all registers up to and including the last register in the series you intend to change. You may leave off values for any registers beyond that.

As an example, say you want to change Index Register Y while leaving all other registers intact. First, examine the registers with CTRL-E.

\* ← Press CTRL-E, then RETURN

A=CD X=B1 Y=C3 P=B5 S=F0

\*

(Note that the register contents you see here are just examples.)

To change Index Register Y to hold the hexadecimal quantity 8A without



changing any other registers, type the existing values of the Accumulator and Index Register X, followed by the new value for Index Register Y.

```
*: CD B1 C3 8A
```

To alter any register other than the Accumulator (A), you have to reenter the contents of all registers up to and including the one which you want to alter.

You *must* press CTRL-E (then RETURN) to examine the registers, or the Monitor will assume you want to alter a memory address. Examining the registers tells the Monitor to switch from using the address pointer to a location to alter, and instead directs it to alter the registers themselves.

To illustrate another register alteration, assume you want to change the Stack Pointer (its contents follow the S when examining the registers) to hold the quantity 4B. First, remember to examine the registers:

```
* ←—— Press CTRL-E, then RETURN
```

```
A=FF X=CD Y=81 P=8A S=FO
```

```
*
```

Now enter the current values, in order, for all other registers, and then the new value for the last one:

```
*:FF CD 81 8A 4B
```

Verify that the change is correct by using CTRL-E to examine the registers once again.

## SAVING AND RETRIEVING MEMORY WITH APPLE II PERIPHERALS

The Monitor allows you to use a cassette recorder to save the contents of a block of memory on magnetic tape. You will want to do this if you create high-resolution graphics shapes (see Chapter 6), or if you write assembly language programs which you want to store. With the Apple II Disk Operating System (DOS), you can store memory contents even more quickly and reliably, on disk. In order to save memory on disk you have to temporarily leave the Monitor, using the DOS through BASIC to save or retrieve memory.

### Saving Memory on Cassette Tape

To save memory on tape, use the Monitor's memory write command. You have to supply the Monitor with the beginning and ending addresses of the memory which you want to save. The command to write memory to the tape begins with the starting address you want to save followed immediately by a period, the ending address of memory which you want to save, and finally the letter W.

For example, the command:

```
*2200.2FFFW
```

tells the Monitor to write the contents of memory, starting at hexadecimal address 2000 and ending with address 2FFF, on the cassette recorder.

The memory write command does not check data it sends out the cassette port; it also does not (and cannot) check for the actual presence of an operating tape recorder connected to the cassette ports. You should do all you can to ensure that the tape you use is free from jamming, dropouts, and other problems which are inherent in using tape cassettes.

When you enter the command to write memory, don't press RETURN until you put the cassette recorder in RECORD mode and the tape is visibly moving in the machine. If the cassette you are using is at the beginning of the tape, let it run for at least five seconds before pressing RETURN, to allow the nonmagnetic leader tape to pass through the recorder.

When you press RETURN, the computer waits ten seconds before sending data. This allows the cassette recorder to erase any previous information (music, voice, or machine-readable data) on the tape. The computer sends a reference tone to the recorder during this time. The Monitor uses this tone later as a locking-on signal in the memory read command (covered in the next section).

When the memory write command finishes, the Apple II speaker beeps once and the Monitor prompt returns.

The memory write command enables you to record from one byte to 64 kilobytes (65,536 bytes) on tape. The Monitor sends data through the cassette output port at an approximate rate of 210 characters per second (based on a 16,384-byte move in 77.5 seconds, after the reference tone). After transmitting the last byte of data, the Monitor sends a checksum byte to the cassette recorder. The memory read command uses this checksum byte to test incoming data for validity.

### Retrieving Data from Cassette Tape

The memory read command enables you to retrieve data from the cassette recorder and load it into memory. To perform the memory read command, enter the starting address (where data from the cassette tape should begin loading into memory) followed immediately by a period, the ending address (where the last byte of data read from cassette will go in memory), and lastly the letter R.

For example, the command:

```
*2000.20FFR
```

means read data from cassette tape into memory, starting at hexadecimal address 2000 and ending at 20FF.

Unlike the write command, the memory read command forces the Monitor to wait until you press the PLAY button on the cassette recorder. The computer waits for the signal tone from the cassette recorder, and the Monitor will lock out the computer until it encounters the signal. Before you press PLAY on the cassette recorder, make sure you position the tape to where the reference tone begins. You can tell the difference between the reference tone and actual data on the tape by listening to it. Remove the connector from the earphone jack to listen to the tape using the cassette recorder speaker. The reference tone is a steady, medium-

pitched hum. Actual data sounds like random noise or static.

Be sure to adjust the cassette recorder playback volume before using the memory read command. The procedure for adjusting the volume is explained in Chapter 2.

The memory read command expects to read in exactly as much memory as you saved using the memory write command. If you write 1024 bytes from memory onto tape and want to retrieve only the first 256 bytes which you recorded, the Monitor will transfer data but you will probably receive an error message. The same is true for reading more data into memory than was recorded on tape; an error message is a strong possibility.

### **Error Conditions in the Memory Read Command**

The Monitor listens to the cassette recorder for at least 3.5 seconds before expecting data from the cassette input port. This allows the Monitor to lock on to the frequency of the reference tone. If the tape contains less than 3.5 seconds of this tone the Monitor will lose the beginning of the data transmission from the cassette, resulting in a checksum error. Furthermore, you will not be sure where on the tape the Monitor began reading data, because the Monitor always attempts to move data into memory from cassette — valid or not. In this case the Monitor error message occurs (a beep from the Apple II speaker followed by the message ERR and the Monitor prompt). To correct the error, rewind the tape to the beginning of the tone. Press PLAY with the Apple II cassette connector disconnected from the earphone jack and time the tone. If it is less than 3.5 seconds before you hear the data transmission, you have to write memory to the tape again. It is probable in this case that you forgot to move past the nonmagnetic leader tape at the beginning of the cassette before recording.

Reading more or less data into memory from cassette than was originally saved on the tape will probably cause an error message to display on the Apple II's screen. The last byte sent to the cassette recorder in the memory write command is a checksum byte. Its value depends on how much data was written as well as what data was written. When a memory read is not equal in length to the original memory write, the Monitor cannot predict which piece of incoming data will be the checksum byte. The Monitor assumes that the last byte it reads from cassette into memory (determined by the ending address in the memory read command) will be followed by a checksum byte. The Monitor performs its own internal checksum calculations on incoming data during the memory read operation and compares this with the checksum byte it reads in from cassette. If the two bytes do not agree, the Monitor displays its error message. It is possible for the checksums to agree by coincidence. As a general rule, you should only read in as much memory as you wrote to the cassette in the first place. This lets the Monitor perform meaningful error checking. Later in this chapter you will see how to verify a memory read operation using the Monitor.

## Saving Memory on Disk

With the Apple II Disk Operating System, saving memory is much quicker and far more reliable than with cassettes. The rules for saving memory on disk are slightly different because you use BASIC rather than the Monitor to perform this function. Even so, this is a machine-level command which is superior to cassette commands included in the Monitor, and you should use it where possible. Before reading this section, you should be familiar with DOS (see Chapter 5). DOS must already be in memory before you begin (see Chapter 2).

If you are in the Monitor, you need to leave it temporarily and get into BASIC under DOS with the CTRL-B or CTRL-C command if you have the Autostart Monitor, or the 3DOG command if you have the standard Monitor.

Here is an example of the DOS command to save memory on disk:

```
BSAVE SHPTABLE, A$3000, L256, S6, D1, V201
```

This command will create a DOS file on disk called SHPTABLE. The next parameter, A\$3000, will save memory on disk starting with address 3000 hexadecimal. The A stands for address. The third piece of information, L256, specifies the length (L) of the memory write, in this case the decimal number 256. The maximum length is 32767 decimal (\$7FFF hexadecimal). BSAVE allows you to use hexadecimal or decimal numbers as address (A) and length (L) parameters. If you use hexadecimal constants in the BSAVE command, make sure to precede the constant with a dollar sign.

The last three parameters in this sample statement (S6, D1, V201) are optional. They specify which disk to use. Use the S (slot) parameter only if you have more than one disk controller card and the disk drive you want to save memory on uses a controller card other than the one last used (usually slot 6, the standard disk drive slot). The D (drive number) parameter is useful but not necessary; specify the drive number only if it differs from the drive you accessed last. The optional V (volume number) parameter is the volume number stored on the directory of the disk you are saving memory on. If the volume number you specify in the BSAVE statement differs from the volume number on the disk you want to save memory on, an error message appears.

## Retrieving Memory from Disk

As with BSAVE, you use DOS to retrieve data from disk and load it into memory. The BLOAD command does this.

Here is an example of BLOAD:

```
BLOAD SHPTABLE, A$3000
```

This example loads the file named SHPTABLE from the disk currently in use and puts the data directly into memory, starting with address 3000 hexadecimal. This command statement also accepts a decimal address. The A (starting address) parameter is not necessary if the file contents start at the same location they were

saved from. You need only specify an address if it differs from the BSAVE starting address. The length parameter (L) is optional but not necessary. DOS checks the length of the disk file itself and terminates the memory read command automatically. The optional disk specification parameters used in BSAVE (S, D, and V) can also be used in BLOAD to indicate a slot, drive, and volume number.

Be careful not to use this command in areas of memory which may be occupied by DOS, the Applesoft interpreter, text or graphics pages, or BASIC variables. It is possible to clobber these areas of memory, losing data which you may have wanted to keep.

### **MOVING AND COMPARING BLOCKS OF MEMORY**

If you read or write memory using tape or disk, the Monitor has two functions which may help you. The move function copies a block of memory into a different range of addresses. The compare function checks two blocks of memory against each other and reports any discrepancies between the two blocks. When used with the Monitor's memory read and memory write commands, these functions furnish you with added insurance that the files you write contain correct data.

#### **The Move Memory Command**

In order to move data from one address in memory to another, you have to supply the destination start address (where you want to move memory to), the source start address (where to move memory from), and the source end address (the last address you want moved). The format of this command is the destination start address followed by the less-than symbol (<), the source start address, a period, the source end address, and the letter M (for move). As with other Monitor commands, all addresses are hexadecimal numbers.

For example, the command:

```
*1200<2000.2100M
```

moves data to address 1200 hexadecimal (the destination start address) from the block starting at 2000 hexadecimal (the source start address) and ending with 2100 (the source end address). The Monitor copies the contents of memory, from addresses 2000 through 2100 to addresses 1200 through 1300 in this example. Since the addresses are hexadecimal, the implied length of the move is 257 bytes decimal, or 101 bytes hexadecimal. The original contents of addresses 2000 through 2100 remain undisturbed.

When you specify addresses in the move memory command, the source start address should be greater than or equal to the source end address. If the source end address is less than the source start address, the Monitor will only move one byte from the source start address to the destination start address and then will terminate the move memory operation.

## Filling Memory

The move memory command also *fills* memory. Filling is the process of moving one or more bytes of data repeatedly to consecutive addresses. Suppose you want to place zeros in a block of memory, starting with address 1D00 and ending with 1DFF. By creative use of the alter memory and move memory commands you can set a block of memory to a predefined set of values, or pattern.

To begin, you might place zeros in the first byte of memory:

```
*1D00:00
```

This is the first step of the fill memory procedure. The second step uses the move memory command to copy the contents of one (or more) bytes into an adjoining block of memory.

Specify a destination start address which is one greater than the last byte of the pattern (in this example it would be 1D01). Set the source start address to the beginning of the pattern (1D00 in this case), and set the source end address to the last byte which you want filled (1DFF) minus the length of the pattern you want to fill memory with (1DFE).

Continuing the example, the command:

```
*1D01<1D00.1DFEM
```

fills locations 1D01 through 1DFF with zeros (or more precisely, with the contents of location 1D00). This procedure only works when the fill pattern exists at the beginning of the block you want to fill. Here's what happens. Since the destination start address comes one byte after the source address, the Monitor moves data at address 1D00 to 1D01 first, moving 00 into this address. When the second byte is moved, the Monitor takes the contents of location 1D01 and moves it into address 1D02. This process continues until the contents of address 1DFE (set to 00 in the last byte transferred) move into address 1DFF. By examining these locations you can see that addresses 1D00 through 1DFF are indeed filled with zeros.

You may want to fill memory with a pattern which is more than one byte long. For example, to move 00 5E 7F FF and fill memory from 1D00 through 1DFF with this four-byte pattern you need to alter four bytes of memory starting at address 1D00:

```
*1D00:00 5E 7F FF
```

The pattern is now in place. To fill memory up to 1DFF with this pattern, enter the command:

```
*1D04<1D00.1DFBM
```

Note that the destination start address occurs one byte after the end of the pattern, the source start address points to the beginning of the fill pattern and the source end address points to the last address to fill minus the length of the pattern:

```
1DFF
-04   (Hexadecimal Arithmetic)
1DFB
```

Once again, if you try this example you can examine memory from 1D00 through 1DFF to verify that the pattern repeatedly occurs in this block.

### The Verify Memory Command

This Monitor command compares two blocks of memory against each other, noting differences between the first block and the second. You can use this command in conjunction with the memory read and memory write commands which the Monitor and the Disk Operating System (DOS) support. If you save memory onto a peripheral device and want to be sure it was written correctly, you can use the verify memory command to do that too.

The format for this command is nearly the same as for the move memory command. Enter the destination start address (where to start comparing memory) followed immediately by the less-than sign (<), the source start address (where to start comparing to the destination), a period, and the source end address (where the last byte of the verify memory operation will take place). The last item on the command line is the letter V, for verify.

Here is an example:

```
*32D0<0.CV
```

This instructs the Monitor to start comparing data at address 32D0 against address 0, and to continue the comparison until address 32DC is compared and verified against address 000C. Notice that no leading zero digits are needed for addresses in a Monitor command line.

If the Monitor encounters a byte in the source block which is not the same as its counterpart in the destination block, the source address displays with its stored value, along with the value it found at the same relative address in the destination block.

For example, if you moved memory from addresses 0000 through 000C to addresses 32D0 through 32DC:

```
*32D0<0.CM
```

and then displayed the source and destination blocks:

```
*0.C
```

```
0000- 4C 3C D4 4C 3A DB 8C 8C
```

```
0008- FF FF 4C 99 E1
```

```
*32D0.32DC
```

```
32D0- 4C 3C D4 4C 3A DB 8C 8C
```

```
32D8- FF FF 4C 99 E1
```

```
*
```

you could visually check the move memory operation. If you alter address 32D8 from its present value (FF) to 5A:

```
*32D8:5A
```

and then enter the verify memory command:

```
*32D0<0.CV
```

the Monitor will compare the source block against the destination block byte-by-byte until it compares data stored at 0008 against the value in 32D8. Since

address 32D8 was just altered, the Monitor displays a discrepancy:

0008-FF (5A)

\*

meaning that the value at address 0008 in the source block does not agree with the same relative address (32D8) in the destination block. The Monitor first displays the value it finds at the source address (0008, the contents of which are FF) and then displays in parentheses the value stored at the destination address where the discrepancy occurred (5A).

The address in the destination block does not appear; the Monitor assumes that you can add numbers in base 16 easily to find the address of the discrepancy in the source block. One way to avoid calculating the address yourself is to switch source and destination blocks:

\*0<32D0,32DCV

which will produce:

32D8-5A (FF)

\*

This message shows that source address 32D8 contains 5A hexadecimal, while its relative counterpart (address 0008 as seen above) contains FF. This method eliminates one calculation but it adds another; that is, you need to calculate a new source end address (32DC).

### Verifying Memory Stored on Apple II Peripherals

The verify memory command is especially useful if you save memory contents onto cassette tape or disk. By saving a portion of memory and then loading it back in at a different location, you can verify that memory was saved properly. The following example shows how to perform this procedure for assembly language programs, shape tables, and other information which you can store from memory to Apple II peripherals.

If you are using the cassette recorder to save memory onto tape, the first step is to enter the memory write command as shown in this example:

\*2000,20FFW

This command writes data from memory, starting at address 2000 and ending with address 20FF, onto cassette tape. Don't forget to start the recorder in RECORD mode *before* you press RETURN. Once the Apple II speaker emits a beep to notify you that the memory write operation is finished, stop the cassette recorder and rewind the tape to where the reference tone begins. Assuming that memory from 2100 to 21FF is available, use the memory read command to load data from cassette to address 2100, ending with address 21FF:

\*2100,21FFR

Don't forget to start the recorder with the PLAY button. When the Apple II speaker emits a beep, the memory read operation is complete. Now you can verify memory against what you saved on tape:

\*2000<2100,21FFV



The verify memory command compares memory, from 2000 to 20FF, to memory which was written to cassette and read back in starting at address 2100. If no discrepancies display, you can be sure that the memory write operation was successful.

To verify memory saved on disk, the same general procedures apply. Before attempting to save or retrieve memory using the disk drives, the Disk Operating System has to be in memory as well. Here, the first step is to BSAVE the block of memory onto disk:

```
BSAVE MEMDATA, A$2000,L$FF
```

Memory saved on disk, in the file named MEMDATA, can be read back in using BLOAD:

```
BLOAD MEMDATA, A$2100
```

Note that the address parameter in this statement is 256 bytes (decimal) higher in memory than the original block saved. When file MEMDATA is read in, the verify memory command compares the two blocks:

```
CALL -151
```

```
*2000<2100.21FFV
```

If no discrepancies occur, you can be sure that memory was properly saved onto disk; otherwise, the Monitor will point out differences between the blocks in order for you to correct them.

## THE GO COMMAND

The Monitor has a command which transfers control of the Apple II to a program at an address you specify. Toward the beginning of this chapter, you saw how to leave the Monitor and go back to the disk-based Applesoft. The command:

```
*3DOG
```

instructs the Monitor to jump to address 3D0 in memory and execute the machine language instruction it finds there. The letter G at the end of the command line stands for GO. If you enter the command shown above and DOS is in memory, address 3D0 contains the first part of the assembly language instruction:

```
JMP $9DB9
```

When the Monitor transfers control to the instruction at address 3D0, the computer branches to address 9DB9, where the DOS routines begin.

The general format for the GO command is the address to transfer control to, followed by the letter G. The address is optional; if none is entered, the Monitor uses its memory pointer as the assumed address.

## USING THE PRINTER

If your Apple II connects to a printer via the Serial Interface or Communications card, you can use the printer for output. To divert all output from the screen to a

printer, enter the slot number of the interface card which controls the printer, followed by `CTRL-P` and `RETURN`. Once you enter this command, all output normally displaying on the screen will be routed to the printer. To select the Apple II screen as the console output device, use slot number 0 with the `CTRL-P` command.

When using this command, be sure that the slot you select has an interface card in it. If no interface card exists at the slot you specify, the Apple II will lock up. The only way to recover from this condition is to press `RESET`.

The printer command works exactly the same way as the `PR#` (slot number) command in BASIC. Both of these commands set the two-byte CSW switch (character output switch) at address 54 (\$36). These two bytes contain an address which points to the character-output subroutine currently in use. By changing the slot with `CTRL-P`, you change the contents of the two-byte CSW switch.

## THE KEYBOARD COMMAND

This command directs the Monitor to accept input from a device other than the Apple II keyboard. As with the printer command, you specify the slot number for the device. Follow the slot number with `CTRL-K`, then press `RETURN`. To return control to the Apple II keyboard, enter a keyboard command with a slot number of 0.

This command sets the KSW (keyboard input switch) at address 56 (\$38) to a two-byte address derived from the slot number entered in the keyboard command.

## SETTING DISPLAY MODES

To view Monitor output on the screen in inverse video, enter the inverse video command, abbreviated as `I`. This will cause all data which the Apple II displays to appear as black letters on a white background. However, any Monitor commands you enter will still display in normal, white-on-black video.

To terminate inverse video, enter the normal video command, abbreviated as `N`. Neither of these commands needs any additional parameter other than the letter `I` or `N`.

## EIGHT-BIT BINARY ARITHMETIC USING THE MONITOR

The monitor performs eight-bit binary addition and subtraction. The results of the arithmetic are also eight bits long. To perform addition, enter a hexadecimal addend followed by a plus sign (+) and a hexadecimal augend. If the result is greater than FF, the Monitor truncates the most significant digit and displays the low-order eight bits of the result, as shown in this example:

```
*7F+8A
=09
```

To perform subtraction, enter the minuend followed by a minus sign (–) and the subtrahend. As with addition, both numbers have to be hexadecimal. If the result is less than zero, the Monitor displays the ones complement result, as shown below:

```
*0A-2D
=DD
```

## USER-DEFINABLE MONITOR COMMAND

By entering CTRL-Y in response to a Monitor prompt, you invoke a special user-definable command. The Monitor automatically jumps to hexadecimal address 3F8 when CTRL-Y is entered. There is enough room at location 3F8 for one machine language jump instruction. If you have a special machine language program somewhere in memory, CTRL-Y could initiate a jump to it via location 3F8.

The example below shows how to set up CTRL-Y to restart disk-based Applesoft without typing the familiar 3DOG command.

First, you need to know the format of a machine language jump instruction. It takes three bytes. The first byte is the instruction code, 4C. The next two bytes are the address to jump to, with a twist. You must specify the *last* byte of the address first (kind of like last name first).

Here's a memory alteration command that sets up a jump instruction to address 3D0:

```
*3F8: 4C D0 03
```

Now try CTRL-Y. That certainly beats 3DOG!

For another example, let's see how you would use CTRL-Y to jump to the Mini-Assembler, which the next section covers in detail. If you enter the Monitor command:

```
*F666G
```

the Mini-Assembler prompt appears:

```
!
```

The address where the Mini-Assembler starts, F666, can be used in a JMP instruction at address 3F8:

```
!3F8: JMP $F666
```

```
03F8- 4C 66 F6    JMP    $F666 ← The Mini-Assembler
!                                     displays this line
```

Although the Mini-Assembler is not discussed until the next section, the line above sets the address of this instruction to 3F8 hexadecimal, and the operand (\$F666) indicates a jump to the beginning of the Mini-Assembler program. To return to the Monitor, enter the command:

```
!$FF69G
```

```
*
```

The Monitor prompt reappears in the lower lefthand corner of the screen. After this point, if you press CTRL-Y the Mini-Assembler prompt appears. Setting the user-definable command to the Mini-Assembler saves keystrokes used in invoking it. Reset the user-definable command by placing a different jump instruction at address 3F8.

## THE MINI-ASSEMBLER

If you have the standard Apple II (or the Integer BASIC card with the Apple II Plus), you have a program in ROM which spares the machine language programmer the torture of hand assembly. The Mini-Assembler resides with Integer BASIC in ROM. It is called mini because the programmer has to use literal addresses, rather than mnemonic labels, as operands in assembly language statements. Also, each line of code you enter is automatically and immediately assembled into machine language. The principal problem here is that you cannot insert or delete instructions at will, as you can with a full-fledged assembler with a text editor.

The principal advantage of the Mini-Assembler is the ability to enter machine instructions directly into the Apple II, while still keeping the convenience of assembly language mnemonic instructions.

The balance of this chapter describes the Mini-Assembler and tells how to use it. The chapter most definitely does not explain assembly language programming concepts. Nor does the chapter cover the 6502 instruction set which is the assembly language the Apple II uses.

So if you're already confused with this talk of assembly, operands, mnemonics, and so forth, stop right now. Learn assembly language programming techniques and the 6502 instruction set first. Then finish reading this chapter.

### ACCESSING THE MINI-ASSEMBLER

The entry point address of the Mini-Assembler program is F666 hexadecimal. To begin from the Monitor, enter the command:

`*F666G`

This will cause the Monitor to jump to the Mini-Assembler. From Integer BASIC or Applesoft (disk or cassette version), enter the immediate-mode command:

`CALL -2458`

When you first invoke the Mini-Assembler, the onboard speaker beeps once. The prompt character for the Mini-Assembler is an exclamation point (!).

### Entry Errors

The Mini-Assembler detects errors which you make when you enter an assembly language instruction. It displays the error by beeping the speaker once and redisplaying the instruction with a caret ( ^ ) under the first incorrect character in the instruction. The location counter does not increment; it remains unchanged since the last character entered so you can reenter the instruction properly.

### MONITOR COMMANDS IN THE MINI-ASSEMBLER

At any time you are in the Mini-Assembler, you can execute Monitor commands.

Immediately after the Mini-Assembler prompt (!), enter a dollar sign (\$), followed by the Monitor command. The example below shows how to examine memory contents from the Mini-Assembler.

```
!$1CFF
1CFF- E6
!
```

This feature saves you the time spent switching back and forth between the Mini-Assembler and Monitor. You can enter any Monitor command while in the Mini-Assembler just by entering the dollar sign as the first character of input. In fact, you will use this feature when you leave the Mini-Assembler.

### LEAVING THE MINI-ASSEMBLER

To leave the Mini-Assembler, use one of the Monitor commands, with the dollar sign prefix. To get back to the Monitor, branch to address FF69 with the \$FF69G command.

\$CTRL-B or \$CTRL-C will put you in BASIC unless you're using disk-based or cassette-based Applesoft. For disk-based Applesoft, use \$3DOG; use \$OG for cassette-based Applesoft.

### INSTRUCTION FORMATS

Although it is not the object of this section to teach you about programming in assembly language, there are some aspects of the Mini-Assembler which you should be aware of before using it. First, the Mini-Assembler maintains an instruction pointer separate from the Monitor's memory pointer. You need to set this pointer before entering instructions. Second, there are various instruction formats used in programming the 6502 microprocessor. These formats depend largely on the addressing scheme used.

The 6502 microprocessor has eleven addressing modes, but only six separate instruction formats. They are described below.

The first, absolute or direct addressing, only requires the one- or two-byte memory address of the operand. For example:

```
AND $303A
```

The Mini-Assembler does not require a dollar sign (\$) before hexadecimal addresses; it assumes that all addresses used are in base 16.

The second addressing format is the immediate addressing mode, as in this example:

```
LDA ##04
```

Note the pound sign (#), the first character of the operand. This is an explicit indicator that the value 04 will load into the Accumulator. Without the pound sign, the Mini-Assembler interprets the instruction as "take the contents of memory location 0004 and load them into the Accumulator," which is actually a default to absolute addressing.

Note also the confusing use of the term *immediate*. Do not confuse immediate addressing in assembly language with immediate execution in BASIC. The actions are quite different. The terms are commonplace so we will use them in spite of the ambiguity.

The third addressing format is the indexed method, which looks like this:

```
CMP $23, X
or
AND $80, Y
```

Both of these instructions are similar in that the X or Y register appears as a second operand. Basically, this format generates a machine language instruction to add the contents of the X or Y register to the address in the first operand, and to use this sum as the address which the instruction references.

Next, the pre-indexed indirect format, as in this example:

```
AND ($F0, X)
```

indicates that the sum of the address (\$F0) and the register (X) contents point to an address in the first 256 bytes of memory which, in turn, contains another address which points to the data to be used as the operand in the instruction.

Post-indexed indirect addressing takes the following form:

```
ORA ($22), Y
```

This instruction format uses the first operand as a pointer to a two-byte address, located in this case at memory location \$22. This instruction adds the contents of the Y register to the address found at \$22; the data at the derived address is used in the machine language instruction. The Mini-Assembler recognizes post-indexed indirect addressing when the first operand is in parentheses, as in the example above.

Indirect addressing is a bit more straightforward than indexed addressing. Here's an example:

```
JMP ($22FE)
```

Here, the JMP instruction does not load address \$22FE into the program counter. Instead, a two-byte address at location \$22FE loads into the program counter. Therefore, the operand in the indirect format is actually a pointer rather than a literal address.

## USING THE MINI-ASSEMBLER

As mentioned in the previous section, the Mini-Assembler maintains a location counter which increments by the length of each assembly language statement you enter. In other words, once the statement you enter is assembled into machine language (every time you enter an instruction), the Mini-Assembler calculates the length of the machine language instruction (1, 2, or 3 bytes) and increments the location counter for the next line.

The first step in using the Mini-Assembler is to set the location counter. Do this as part of the first assembly language statement you enter. For example:

```
!8DB0:LDA #$04
```

Directly after the Mini-Assembler prompt, enter the base address for the assembly language code you are entering (in this case, 8DB0), followed by a colon (:) and the first assembly language statement. You do not have to enter a new location counter value for the next instruction. The Mini-Assembler calculates the next address unless you reset the location counter by entering another address as shown above.

Once you set the location counter, enter subsequent assembly language instructions, one per line. After the first line, enter a blank followed by the next assembly language statement, like this:

```
! JSR FB1E
```

This directs the Mini-Assembler to calculate the new location counter value.

### A Sample Session

This section explains Mini-Assembler operation in step-by-step detail. The object of this sample session is to create a small program which uses the Apple II game control inputs and the onboard speaker to create sounds. The procedure for this program is to read values from paddle 0 and paddle 1, using the built-in Monitor subroutine PREAD (at address FB1E). The value of paddle 0 is the interval between clicking the speaker (0=shortest delay, FF=longest delay), and the value of paddle 1 is another interval, related inversely to paddle 0 (0=longest interval, FF=shortest interval).

The program will begin at 1D00, and will use address 1CFF for storing the reading from paddle 0.

When you enter each line of the assembly language program, the Mini-Assembler overlays the line you entered with the current location counter value, operation code, and operand in machine language form (also known as *object code*), along with the instruction mnemonic you entered. For example:

```
1D00- A2 00 LDX ##00
```

The location counter displays at the beginning of the assembled line, followed by a dash. After this field, the operation code (A2 for this LDX instruction) displays, followed by the last byte of the instruction. In the case of three-byte instructions (those which refer to a two-byte address), the low-order byte appears before the high-order byte. Lastly, the instruction mnemonic appears.

The annotated sample session appears below. Note that each line produced by the Mini-Assembler appears here below the line you enter to generate it.

```
!1D00:LDX ##00      ← Set location counter and enter first instruction

1D00-  A2 00      LDX  ##00
! JSR FB1E        ← All numbers are hexadecimal ($ prefix unnecessary)

1D02-  20 1E FB   JSR  $FB1E
! STY 1CFF

1D05-  8C FF 1C   STY  $1CFF
! INX
```

```

1D08-   E8           INX
! JSR FB1E

1D09-   20 1E FB     JSR   $FB1E
! LDA C030

1D0C-   AD 30 C0     LDA   $C030
! DEC 1CFF

1D0F-   CE FF 1C     DEC   $1CFF
! BNE 1D0C ← Mini-Assembler computes the relative jump (F8)

1D12-   D0 F8       BNE   $1D0C
! LDA C030

1D14-   AD 30 C0     LDA   $C030
! INY

1D17-   C8           INY
! BNE 1D14

1D18-   D0 FA       BNE   $1D14
! JMP 1D00

1D1A-   4C 00 1D     JMP   $1D00
!
```

After entering this program, you should check it for accuracy. The best way to do this is to list the program in memory, preferably in assembly language format. But you will need to use the Monitor to do so as described below.

As an additional safeguard, you may want to save the program on cassette (use the Monitor W command) or disk (with the BASIC BSAVE statement).

To run the program, branch to location 1D00. Use the G command in the Monitor, or CALL 7424 from BASIC. Fiddle with the game controls and see how they affect the speaker. To end the program, press RESET.

## DISASSEMBLED LISTINGS

The Monitor contains a command which you can use to list machine language instructions in assembly language format, even if your Apple II does not have the Mini-Assembler in ROM. The command L, for List, disassembles 20 machine language instructions into assembly language statements and displays them on the screen or other output device you select. The List command uses the location counter as a pointer to the next instruction to disassemble. Therefore, if you just enter L after entering the program above, disassembly will start with address 1D1D, and it will not list any of the program you entered.

It is good practice to set the location counter when using the List command. Here is a disassembled listing of the sound program:

```

!$1D00L

1D00-   A2 00       LDX   #$00
```



1D02-	20 1E FB	JSR	\$FB1E
1D05-	8C FF 1C	STY	\$1CFF
1D08-	E8	INX	
1D09-	20 1E FB	JSR	\$FB1E
1D0C-	AD 30 C0	LDA	\$C030
1D0F-	CE FF 1C	DEC	\$1CFF
1D12-	D0 F8	BNE	\$1D0C
1D14-	AD 30 C0	LDA	\$C030
1D17-	C8	INY	
1D18-	D0 FA	BNE	\$1D14
1D1A-	4C 00 1D	JMP	\$1D00
1D1D-	9F	???	
1D1E-	4E A5 12	LSR	\$12A5
1D21-	A4 96	LDY	\$96
1D23-	A3	???	
1D24-	D0 A4	BNE	\$1CCA
1D26-	EF	???	
1D27-	A4 62	LDY	\$62
1D29-	A2 70	LDX	##70

In this case, the last eight disassembled instructions are immaterial, since the program ends at address 1D1A.

Note that the List command is a Monitor facility, independent of the Mini-Assembler (hence the dollar sign prefixing the command). By entering L (\$L in the Mini-Assembler) and pressing RETURN without setting the location counter, you direct the Monitor to disassemble the next 20 instructions it finds after those just listed.

## TESTING AND DEBUGGING PROGRAMS

Together with the Mini-Assembler, the Monitor in the standard Apple II furnishes you with debugging features which are very helpful when you are programming in assembly language. Programs in low-level languages are probably the hardest to debug and test. Instead of simply printing contents of variables, you must inspect memory locations, registers, and the program itself. Two Monitor commands, Step and Trace, are used for this purpose.

The Step and Trace commands are not available on versions of the Apple II with the Autostart Monitor.

### The Step Command

While it is easy enough to try testing an assembly language program by witnessing the symptoms of its operation, this is frequently not a very efficient or effective method of isolating errors. If the program is small enough, it is possible to run it step by step, checking the results of each machine language instruction after the Apple II performs it. The Step command does just that.

When you execute the Step command, the Monitor disassembles and displays the instruction pointed to by the location counter, executes it, displays the con-

tents of the microprocessor registers, and returns control of the Apple II to the Monitor.

The format of the Step command is an optional address parameter (to set the location counter) followed by the letter S.

The Step command below performs the first instruction in the sample sound program. The contents of the X register display as 0. The other three registers whose contents display are unchanged.

```
*1D00S
1D00-  A2 00      LDX  #$00
      A=FF X=00 Y=8C P=32 S=F8
*
```

You can alternate between the Step command and other monitor commands (such as examine memory). To see this in action, step through the sample sound program until you get to the STY instruction at memory location 1D05. You will have to enter the S command nine times, since the JSR instruction at location 1D02 calls a subroutine at location FB1E and you must step through it before you get back to 1D05. Once you're there, use the examine memory Monitor command to check location 1CFF. The instruction at address 1D05 stores the reading of paddle 0 at address 1CFF.

```
*S
1D05-  8C FF 1C    STY  $1CFF
      A=00 X=00 Y=00 P=32 S=F8
*1CFF
1CFF-  00
*
```

As you can see from the memory examine command, the contents of the Y register (3F) are now stored at address 1CFF. You can intersperse most Monitor commands with the Step command.

### The Trace Command

Sometimes a program may be too long to allow you to execute each instruction step by step. Instead, you may want to see each program step execute, but only interrupt the execution of the program when it is necessary to do so. The Monitor's Trace command performs this function. Its output is similar to the Step command, except that it saves you the effort of entering a Step command for each instruction the Apple II executes. In order to stop the Trace command, you can press the **RESET** key or imbed a **BRK** assembly language instruction in the program; when the Monitor encounters this instruction it returns control of the computer to you (via the Monitor).

The format for the Trace command is an optional address followed by the letter T. Here is the first part of a trace of the sample sound program:

\*1D00T

```

1D00-   A2 00      LDX   #$00
      A=FF X=00 Y=8C P=32 S=F6
1D02-   20 1E FB   JSR   $FB1E
      A=FF X=00 Y=8C P=32 S=F6
FB1E-   AD 70 C0   LDA   $C070
      A=00 X=00 Y=8C P=32 S=F4
FB21-   A0 00      LDY   #$00
      A=00 X=00 Y=00 P=32 S=F4
FB23-   EA        NOP
      A=00 X=00 Y=00 P=32 S=F4
FB24-   EA        NOP
      A=00 X=00 Y=00 P=32 S=F4
FB25-   BD 64 C0   LDA   $C064,X
      A=00 X=00 Y=00 P=32 S=F4
FB28-   10 04      BPL   $FB2E
      A=00 X=00 Y=00 P=32 S=F4
FB2E-   60        RTS
      A=00 X=00 Y=00 P=32 S=F4
1D05-   8C FF 1C   STY   $1CFF
      A=00 X=00 Y=00 P=32 S=F6
1D08-   E8        INX
      A=00 X=01 Y=00 P=30 S=F6
1D09-   20 1E FB   JSR   $FB1E
      A=00 X=01 Y=00 P=30 S=F6
FB1E-   AD 70 C0   LDA   $C070
      A=27 X=01 Y=00 P=30 S=F4
FB21-   A0 00      LDY   #$00
      A=27 X=01 Y=00 P=32 S=F4
FB23-   EA        NOP
      A=27 X=01 Y=00 P=32 S=F4
FB24-   EA        NOP
      A=27 X=01 Y=00 P=32 S=F4
FB25-   BD 64 C0   LDA   $C064,X
      A=27 X=01 Y=00 P=30 S=F4
FB28-   10 04      BPL   $FB2E
      A=27 X=01 Y=00 P=30 S=F4
FB2E-   60        RTS

```

The disadvantage of using the Trace command over the Step command is that you have less control over the execution of each step of the program. When you initially enter the program, you have to put in BRK instructions at key points. These are junctions in the logic of the program. Of course, you can replace these BRK instructions with NOP instructions (no operation code), but this can be distracting once you have finally debugged the program.

Additionally, the Trace command works at a fraction of the speed of the assembly language program. For instance, try substituting the instruction in the sample sound program at address 1D1A with a BRK instruction. By entering the command 1D00G this routine takes a fraction of a second to execute. However, by entering 1D00T, the program takes anywhere from 60 to 70 seconds to execute. Therefore, if you have a large program to test, be conservative with the Trace instruction if you want it to be any help at all.

### More About the Location Counter

As mentioned before in the section on the Step Command, you can use most Monitor commands alternately with the Step and Trace commands. There are some exceptions to this rule, however. The List, Go, and user-defined CTRL-Y commands all modify the location counter when you enter them. This will disrupt the flow of the program you are stepping through or tracing unless you reset the location counter after executing one of these commands. The example below shows how the location counter's value is disrupted by using one of these commands.

```
*1D00S
1D00-   A2 00      LDX   ##00
      A=62 X=00 Y=00 P=32 S=F8
*S
1D02-   20 1E FB   JSR   $FB1E
      A=62 X=00 Y=00 P=32 S=F8
*L                                     ← List command here

FB1E-   AD 70 C0    LDA   $C070
FB21-   A0 00      LDY   ##00
FB23-   EA         NOP
FB24-   EA         NOP
FB25-   BD 64 C0    LDA   $C064, X
FB28-   10 04      BPL   $FB2E
FB2A-   C8         INY
FB2B-   D0 F8      BNE   $FB25
FB2D-   88         DEY
FB2E-   60         RTS
FB2F-   A9 00      LDA   ##00
FB31-   85 48      STA   $48
FB33-   AD 56 C0    LDA   $C056
FB36-   AD 54 C0    LDA   $C054
FB39-   AD 51 C0    LDA   $C051
FB3C-   A9 00      LDA   ##00
FB3E-   F0 0B      BEQ   $FB4B
FB40-   AD 50 C0    LDA   $C050
FB43-   AD 53 C0    LDA   $C053
FB46-   20 36 F8   JSR   $F836
*S                                     ← Changes location counter, so
                                     next step is at $FB49
                                     instead of $FB1E

FB49-   A9 14      LDA   ##14
      A=14 X=00 Y=00 P=30 S=F6
*S
FB4B-   85 22      STA   $22
      A=14 X=00 Y=00 P=30 S=F6
*
```

### Useful Monitor Subroutines

In some cases, BASIC is not powerful enough to perform all of the functions you may need in a program. This, of course, is one reason why programmers resort to

assembly language subroutines for their BASIC programs. This section shows how to reference them from a BASIC program.

By weaving assembly language programs in with a BASIC program, you can create as many problems as you intended to solve. Where in memory are you going to put the assembly language programs? Remember, the Apple II memory contains four large reserved areas (text/low-resolution graphics pages and both high-resolution graphics pages). DOS and the Applesoft interpreter may take up memory too. Locating a program where it will not cause problems is dependent on memory size and the version of Apple II you use.

The Monitor is your best source for assembly language subroutines for three reasons: first, being in ROM, you don't need to worry about relocating code; second, the Monitor routines have already been debugged; and lastly, the intrinsic routines do not take up one byte of additional memory. The useful Monitor subroutines are listed in Appendix D.

### **Incorporating the Subroutine**

If you decide to use a Monitor subroutine in a BASIC program, first be sure that there is no BASIC equivalent for it. This will save you the trouble of making a program more complicated than necessary. Next, check whether the assembly language subroutines need parameters passed from the BASIC program. If you have to set values in the microprocessor registers before executing the subroutine, or if the result of a subroutine resides in a register after execution, you will have to use extra assembly language instructions to interface with BASIC. Most Monitor subroutines need no parameters from BASIC; those which do frequently have a BASIC equivalent anyway.

Once you know which subroutine to use, you may want to document it in a way which makes the meaning clear. For instance, `CALL -936` clears the text screen and places the cursor in the upper lefthand corner of the screen. One way of making the `CALL` statement more descriptive is to set a variable at the beginning of the program, as follows:

```
10 CLSCREEN=-936
```

and to reference it later in the program:

```
1510 CALL CLSCREEN
```

This makes the context of the `CALL` statement clearer to someone who has to read it, but it does add one statement to the program. These finer elements of style will make your program easier to read and debug.

### **Problems to Avoid**

If you have an editor/assembler available for the Apple II, it is easy to relocate programs by resetting the origin point and reassembling. However, if you wrote a machine language subroutine with the Mini-Assembler, and the subroutine is

designed to be used with BASIC, you may run into problems which force you to rewrite the subroutine for versions of the Apple II with different memory sizes. This will happen if you use memory locations used by DOS, the graphics pages, or disk-based or cassette-based Applesoft. Try to use Monitor subroutines wherever possible.

If you program in Applesoft, always use the USR function if you have to pass parameters to and from the subroutine, instead of the CALL statement. Addresses 9D through A3 store the value of the parameter passed by USR, and you can use this area for parameters to pass back to BASIC. Use the POKE statement to put a JMP instruction in locations 10 through 12 (0A through 0C hexadecimal). These locations must contain a JMP instruction to the beginning of the machine language subroutine invoked by USR.

## **INTEGRATING YOUR PROGRAM WITH BASIC**

In BASIC, the statements LOMEM: and HIMEM: protect your assembly language program from being written over by BASIC. There are some requirements in setting up an assembly language program if you also intend to have disk-based or cassette-based Applesoft or the Disk Operating System in memory at the same time. The general procedure for using an assembly language subroutine or program with BASIC and DOS is as follows:

1. Boot DOS.
2. Load the Applesoft interpreter from disk or cassette, if necessary.
3. Set LOMEM: and HIMEM: values.
4. Load the assembly language program.
5. Load the BASIC program from disk or tape (or type it in).

The Disk Operating System resets HIMEM: after you load it from the disk into memory. The Applesoft interpreter resets LOMEM: after it is loaded. You reset LOMEM: or HIMEM: to allow room for your assembly language programs, and then load them into that safe space. Subsequently loading and running a BASIC program affects only the remaining memory between LOMEM: and HIMEM:.

For more help on finding space for your assembly language program, check the memory maps in Appendix G. Also, refer to the discussion of LOMEM: and HIMEM: in Chapter 8.

# 8

## Compendium of BASIC Statements and Functions

This chapter describes the syntax for all Apple II BASIC statements and functions. Statements are described first, listed in alphabetical order; then functions are described, also in alphabetical order.

This chapter serves as a reference for all statements and functions. Chapters 3 through 7 describe programming concepts. They also give examples of statements and functions used in programs.

### IMMEDIATE AND PROGRAMMED MODES

Most statements can be executed in immediate or programmed mode. Unless otherwise stated, you can assume that a statement can be used in both modes. Exceptions are identified. Some statements can be used in one mode, but not the other; other statements can be used in both modes, but only one mode is practical.

Some statements are Disk Operating System (DOS) statements. They can be used as shown in immediate mode. In programmed mode, however, they must be issued as part of a PRINT statement string, the first character of which is CTRL-D (ASCII code 4). Thus, the following two statements are equivalent:

```
!CATALOG  
!PRINT CHR$(4); "CATALOG"
```

Note that instead of using CHR\$(4) as shown above, you can type a quotation mark, followed by CTRL-D, and then another quotation mark. On the screen it will

seem you have only pressed the quotation mark key twice. The CTRL-D character is there even though you can't see it.

## BASIC VERSIONS

All statements and functions are available in both Integer BASIC and Applesoft unless otherwise stated. Where a statement or function works differently in the two versions of BASIC, the differences are noted.

## NOMENCLATURE AND FORMAT CONVENTIONS

We use a standard scheme for presenting the general form of each statement and function. Listed below are the punctuation, capitalization, and other mechanical conventions we use.

{ }	Braces indicate a choice of items. One of the enclosed items must be present; braces do not appear in an actual statement.
[ ]	Brackets indicate that the enclosed parameter is optional; brackets do not appear in an actual statement.
...	Ellipses indicate that the preceding item can be repeated; ellipses do not appear in actual statements.
line numbers	A beginning line number is implied for all programmed mode statements.
other punctuation	All other punctuation marks — commas, semicolons, quotation marks, and parentheses — must appear as shown.
UPPER CASE	Upper-case words and letters must appear exactly as shown.
<i>italics</i>	Generic terms are italicized. The programmer supplies the exact wording or value, according to the generic term definitions listed below.

The following italicized generic terms are used in statement and function definitions. Any italicized terms not listed here are peculiar to the statement in which they appear. They are defined in the text that describes that statement.

<i>col</i>	Low-resolution graphics column number; a numeric expression which has a value between 0 and 39.
<i>colh</i>	High-resolution graphics column number; a numeric expression which has a value between 0 and 279.
<i>const</i>	Any numeric or string constant.
<i>Dn</i>	A disk drive number which must be specified as D0 or D1.



<i>expr</i>	Any numeric string, relational, or Boolean (Applesoft only) constant, variable, or expression; any valid combination thereof.
<i>expr\$</i>	Any string constant, variable, or expression.
<i>exprnm</i>	Any numeric constant, variable, or expression.
<i>filename</i>	Any disk file name.
<i>line</i>	Any BASIC program line number.
<i>line<sub>i</sub></i>	One of many BASIC program line numbers.
<i>memadr</i>	A numeric expression, variable, or constant that evaluates to any memory address. Memory addresses may range from -65535 to 65535 (decimal), where -65535 is the same as 1, -65534 equals 2, etc.
<i>memloc</i>	Any memory location specified by an integer constant between 0 and 65535 (decimal) or \$0 and \$FFFF (hexadecimal). Hexadecimal constants are identified by a dollar sign (\$) prefix.
<i>message</i>	Any text string enclosed in quotes.
<i>row</i>	Low-resolution graphics row number; a numeric expression which has a value between 0 and 47.
<i>rowh</i>	High-resolution graphics row number; a numeric expression which has a value between 0 and 191.
<i>Sn</i>	Slot number for input or output; must be S0, S1, S2, S3, S4, S5, S6, or S7.
<i>var</i>	In Integer BASIC, any numeric or string variable. In Applesoft, any numeric, integer, or string variable.
<i>varnm</i>	Any numeric variable name.
<i>var(sub)</i>	In Integer BASIC, any subscripted numeric variable. In Applesoft, any subscripted integer, numeric, or string variable.
<i>Vn</i>	An identifying disk volume number (between V0 and V255).

## STATEMENTS

### APPEND

Opens a file (see OPEN), and positions the file pointer at the end of the file.

Format:

APPEND *filename* [*Dn*] [*Sn*] [*Vn*]

TABLE 8-1. Machine Language Fix for APPEND

MACHINE LANGUAGE		6502 ASSEMBLY LANGUAGE	
Decimal	Hexadecimal	Instruction	Comments
169	A9	LDA \$0	The Monitor routine at \$FDED outputs the character in register A (\$0 in this case) to the currently selected output device, the disk. See Appendix D.
0	0	JMP \$FDED	
76	4C		
237	ED		
253	FD		

A memory buffer of 595 bytes is allocated for the text file specified. The file must be a sequential file. The WRITE command can now be used to store information on the file, starting at the first unused byte. This will be immediately following the last character in the file unless there are unused bytes in the middle.

Occasionally, APPEND will not start at the first unused byte in the file (often the end of the file). Instead it starts at the beginning of the file. (Horrors!) To make sure this doesn't happen, your program should always write an end-of-file marker before closing a file it has written to. The short machine language subroutine in Table 8-1 does the trick. POKE this into memory anywhere there are five free bytes (locations 768 through 772 are OK unless you're using them for something else). Then call the subroutine (use CALL) just before closing the file.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results. V0 will match any disk. If the file is already open, APPEND closes it and reopens it (see CLOSE).

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

APPEND is a DOS command, requiring PRINT and CTRL-D in programmed mode.

May not be used in immediate mode.

## AUTO

Sets automatic line numbering mode in Integer BASIC.

### Format:

**AUTO** *line* [, *increment*]

Line numbers are automatically displayed each time you key RETURN, starting

with *line*, and increasing each time by *increment*, which defaults to 10 if not specified. Type CTRL-X to erase an automatic line number; automatic line numbering resumes unless MAN is entered on the next line (see MAN).

Can be used only in immediate mode.

Not available in Applesoft.

## BLOAD

Retrieves a binary file from the disk and stores it in the specified section of memory.

### Format:

BLOAD *filename* [,*Amemloc*] [,*Dn*] [,*Sn*] [,*Vn*]

If the A parameter is absent, the specified file is placed in memory beginning at the memory location from which the file was saved (see BSAVE). If the A parameter is present, the file goes into memory at *memloc*.

BLOAD must be used with care. Anything already in the section of memory where it is placed (such as your program, Applesoft, DOS, etc.) will be overwritten.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## BRUN

Retrieves a binary file (which should be a machine language program), stores it in the specified section of memory, and then executes a machine language jump (JMP in 6502 assembly language) to the starting memory location.

### Format:

BRUN *filename* [,*Amemloc*] [,*Dn*] [,*Sn*] [,*Vn*]

If the A parameter is absent, the specified file is placed in memory beginning in the memory location from which the file was saved (see BSAVE). If the A parameter is present, the file goes into memory at *memloc*.

A machine language program may work properly at only one memory location. Check carefully for instructions that are address dependent before loading to a new memory location. BRUN overwrites anything in the section of memory in which it stores the file; this could be awkward if DOS or Applesoft is destroyed in the process.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## BSAVE

Creates a disk file and saves a section of the Apple II's memory on it, in binary.

### Format:

**BSAVE** *filename* ,*Amemloc* ,*Llength* [*Dn*] [*Sn*] [*Vn*]

The A parameter specifies the starting address of the memory section to save. The L parameter specifies the number of bytes to save. *length* must be an integer in the range 0 through 32767 (decimal). It may be either a decimal or hexadecimal constant. Hexadecimal constants are identified by a dollar sign (\$) prefix.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## CALL

Branches to a machine language subroutine at a specified location.

### Format:

**CALL** *memadr*

CALL can be used with subroutines that you write yourself, as well as with various intrinsic subroutines which are listed in Appendix D.

## CATALOG

Displays a list of all files on the specified disk.

### Format:

**CATALOG** [*Dn*] [*Sn*]

CATALOG first prints DISK VOLUME followed by the volume number of the disk. If the volume parameter  $Vn$  is included in the CATALOG command, it is ignored.

The list of files on the disk is displayed below the volume number. For each file, CATALOG prints a code letter indicating the type of file, the number of sectors required to store the file, and the name of the file. An asterisk appears to the left of the file type if the file is locked (see LOCK). The file types and their codes are:

I	Integer BASIC Program
A	Applesoft Program
T	Text File
B	Binary (Machine Language) File

If a file length exceeds 255 sectors, the file length is displayed modulo 255; i.e., 0 is printed if the file length is 256, 1 if it is 257, etc.

$Dn$  and  $Sn$  can be specified in any order. If  $Dn$  or  $Sn$  is absent, the last-referenced drive or slot is used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## CHAIN

Loads and runs an Integer BASIC program from the disk, without clearing the values of any variables or arrays.

### Format:

CHAIN *filename* [ $Dn$ ] [ $Sn$ ] [ $Vn$ ]

The CHAIN command may only be used in Integer BASIC, and may only be used to load an Integer BASIC program.

If the file does not exist on drive  $Dn$  of slot  $Sn$ , the FILE NOT FOUND error message is displayed. If the disk in drive  $Dn$  of slot  $Sn$  is not volume  $Vn$ , the VOLUME MISMATCH error results.

$Dn$ ,  $Sn$ , and  $Vn$  can be specified in any order. If  $Dn$  or  $Sn$  is omitted, the last-referenced drive or slot is used. V0 is used if  $Vn$  is absent. Also,  $n$  can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## CLEAR

This Applesoft statement assigns 0 to all numeric variables and numeric array elements. Also assigns a null value to all string variables and string array elements.

### Format:

CLEAR

Executing this statement is equivalent to turning the Apple II off, then turning it back on and reloading the program into memory. A program will continue to run following CLEAR providing the effects of the CLEAR statements do not adversely affect program logic.

For Integer BASIC, use CLR.

## CLR

This Integer BASIC command assigns 0 to all numeric variables and array elements, and assigns a null value to strings.

### Format:

CLR

Also undimensions all arrays and strings. You can still print array values after executing a CLR statement, as long as no variables have been assigned values in the interim.

CLR can be used only in immediate mode.

For Applesoft, use CLEAR.

## CLOSE

Deallocates the buffer used by the specified disk file, and if the last operation on the file was WRITE, saves anything left in the output buffer on the file.

### Format:

CLOSE [*filename*]

You *must* CLOSE any file you have used the WRITE statement with in order to avoid losing data. If *filename* is present, only that file is closed. If *filename* is not present, all files are closed (except a controlling EXEC file).

Occasionally a sequential file will exactly fill a sector as it is closed. Under these conditions, a subsequent APPEND will occur at the beginning of the file rather than the end. To forestall this, call the short machine language subroutine in Table 8-1 just before the CLOSE statement. You can POKE the subroutine anywhere there are five free bytes (e.g., locations 768 through 772 unless otherwise in use).

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## COLOR=

Sets the color for low-resolution graphics.

### Format:

COLOR= *exprnm*

TABLE 8-2. Low-Resolution Color Codes

Code	Color	Code	Color	Code	Color	Code	Color
0	Black	4	Dark Green	8	Brown	12	Green
1	Magenta	5	Grey	9	Orange	13	Yellow
2	Dark Blue	6	Medium Blue	10	Grey	14	Aqua
3	Purple	7	Light Blue	11	Pink	15	White

Until the next COLOR statement, all PLOT, VLIN, and HLIN statements will be in the color specified. The color codes are listed in Table 8-2. The *exprnm* must have a value in the range 0 to 255; real values are converted to integers. Values greater than 15 repeat the colors shown above (0, 16, 32, etc. are black, and so on). COLOR = 0 if not previously specified.

COLOR has no effect if used while in high-resolution graphics mode. When used while in text mode, COLOR is one factor in determining which character is placed on the screen by a PLOT instruction. For a detailed description of this feature, see PLOT.

**CON**

This Integer BASIC command resumes program execution at the next instruction after a halt.

**Format:**

CON

CON operates after execution has been halted by CTRL-C, and sometimes after RESET. If there is no interrupted program, CON simply locks up the system. A program cannot be continued after it is interrupted by CTRL-C during an INPUT statement.

If a program line has been changed or added, or an error message generated since program execution was halted, CON will work sometimes, but may produce an error message or lock up the system.

CON can be used only in immediate mode.

For Applesoft, see CONT.

**CONT**

This Applesoft command resumes execution at the next instruction after a halt.

**Format:**

CONT

CONT operates after execution has been halted by STOP, END, or CTRL-C. If an INPUT statement is interrupted by CTRL-C, the program cannot be continued. If there is no interrupted program, or a program line has been changed or added, or an error message generated since program execution was halted, CONT will produce the message ?CAN'T CONTINUE ERROR.

For Integer BASIC, see CON.

## DATA

Creates a list of values to be assigned by READ statements in Applesoft.

### Format:

**DATA *const* [ ,*const* . . . ]**

DATA statements may appear anywhere in a program; they need not be executed to be accessed by a READ command.

The DATA statement specifies either numeric or string values. String constants are usually enclosed in quotation marks; the quotes are not necessary unless the string contains blanks (spaces), commas, or colons. A quotation mark cannot be represented in a *const*; it must be specified using a CHR\$(34) function.

One or more of the *const* parameters can be null, i.e., nothing but blanks. A null *const* is assigned as zero to a numeric variable, or a null string ("" ) to a string variable.

You will receive no error message if you enter a DATA statement in immediate mode, but the elements will not be accessible to a READ command.

Not available in Integer BASIC.

## DEF FN

The DEF FN statement allows special purpose functions to be defined and used within Applesoft programs.

### Format:

**DEF FN*nvar* (*arg*)=*exprnm***

Real variable *nvar* identifies the function, which is subsequently invoked using the name FN*nvar*.

The function is defined by *exprnm*. *arg* is a dummy variable name which can (and usually does) appear in *exprnm*. Its use in a DEF FN statement has no effect on any other variable with the same name elsewhere in the program.

When FN*nvar* is invoked, the value of the dummy variable *arg* is specified by a numeric expression, variable, or constant. The values of all other variables in *exprnm* must be defined before FN*nvar* is invoked. See also FN in the Functions section of this chapter.



The entire DEF FN statement must appear on a single program line. However, a previously defined function can be included in *exprnm*, so user-defined functions of any desired complexity can be developed. A user-defined function cannot, however, invoke itself directly or indirectly (by invoking a function which eventually invokes it).

The function name *nvar* can be reused, and therefore redefined by another DEF FN statement appearing later in the same program.

Not available in Integer BASIC.

The DEF FN definition statement is illegal in immediate mode. However, a user-defined function that has been defined by executing a DEF FN statement since the last NEW, CLR, or LOAD command can be referenced in an immediate mode statement.

## DEL

Eliminates the specified program lines.

### Format:

DEL *line*<sub>1</sub>, *line*<sub>2</sub>

All program lines greater than or equal to *line*<sub>1</sub> and less than or equal to *line*<sub>2</sub> are removed from the program currently in memory. If *line*<sub>1</sub> does not exist, the deletion starts at the next higher line number. If *line*<sub>2</sub> does not exist, the deletion ends at the next lower line number.

DEL must be followed by two line numbers which are separated by a comma. Neither line number can be negative, and the second line number must be greater than or equal to the first. If the line numbers are identical, one line (at most) is deleted.

DEL may only be used in immediate mode in Integer BASIC.

If DEL is used in programmed mode (possible only in Applesoft), the indicated deletions take place and the program halts. CONT will not continue the program in this case.

## DELETE

Erases a file from the disk.

### Format:

DELETE *filename* [,D*n*] [,S*n*] [,V*n*]

The file with the specified name is removed from the disk.

If the file does not exist on drive D*n* of slot S*n*, the FILE NOT FOUND error message is displayed. If the disk in drive D*n* of slot S*n* is not volume V*n*, the VOLUME MISMATCH error results.

$Dn$ ,  $Sn$ , and  $Vn$  can be specified in any order. If  $Dn$  or  $Sn$  is omitted, the last-referenced drive or slot is used.  $V0$  is used if  $Vn$  is absent. Also,  $n$  can be absent;  $D0$ ,  $S0$ , or  $V0$  will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## DIM

Reserves space in memory for an array or string.

Because of the extensive differences between Integer BASIC and Applesoft, DIM is discussed below as it operates in each language.

### Integer BASIC Format:

```
DIM var (sub) [,var (sub) . . .]
```

Only numeric arrays of one dimension and simple string variables may be dimensioned in Integer BASIC.

When an array is dimensioned, space is set aside in memory for the number of elements equal to *sub* plus 1. They are numbered 0 through *sub*. Element 0 of an array is identical to the simple variable of the same name (e.g.,  $A(0)=A$ ).

DIM statements declare the maximum lengths of string variables. In this case *sub* is the string length.

Every subscript *sub* must be between 1 and 255 in a DIM statement. Aside from this, the maximum allowable dimensions are limited by available memory.

If you reference an array using a subscript greater than the largest subscript declared in the DIM statement for that array, the message \*\*\* RANGE ERR occurs. If you attempt to use more characters in a string than it was dimensioned for, the message \*\*\* STRING ERR is generated.

DIM does not assign the elements of Integer BASIC arrays any particular value when it is executed. Therefore, you must initialize every array (e.g., to zero) after dimensioning it. String variables, on the other hand, always have a null value after first being dimensioned.

### Applesoft Format:

```
DIM var (sub [,sub . . .]) [,var (sub [,sub . . .]) . . .]
```

The DIM statement identifies arrays with one or more dimensions as follows:

<code>var(sub,)</code>	Single-dimension array
<code>var(sub<sub>i</sub>,sub<sub>j</sub>)</code>	Two-dimension array
<code>var(sub<sub>i</sub>,sub<sub>j</sub>,sub<sub>k</sub> . . .)</code>	Multiple-dimension array

Applesoft supports three types of arrays: integer, real, and string. Each element of an array is of the type specified by the variable name for the array. The number of dimensions in an array is determined by the number of subscripts in the DIM statement. When an array is referenced, each subscript must fall within the range

0 through *sub* , where *sub* is the corresponding subscript of the same variable in the DIM statement.

The number of dimensions in an array is limited by the amount of memory available. The maximum number of dimensions an array can ever have is 88, and this is only possible when most of the subscripts are 0. A DIM statement with 89 or more subscripts, or one that otherwise exceeds memory limitations will produce the message ?OUT OF MEMORY ERROR.

If you attempt to use an array with a subscript that is out of range or with the wrong number of subscripts the message ?BAD SUBSCRIPT ERROR will appear.

If an array is referenced before a DIM statement for that array has been executed, Applesoft assigns a default value of 10 to each subscript. The array is thereafter treated just as if a DIM statement with a subscript of 10 for each dimension had been executed.

An array can never be dimensioned twice, even one that has been dimensioned by default. If you attempt to dimension an array that has already been dimensioned, you will be treated to the message ?REDIM'D ARRAY ERROR.

## DRAW

This Applesoft statement draws a high-resolution graphics shape on the screen.

### Format:

DRAW *exprnm* [ AT *colh* , *rowh* ]

The shape identified by the integer value of *exprnm* is drawn in the color determined by the last-executed HCOLOR statement. The scale and rotation of the shape must be set by SCALE and ROT commands before the DRAW is executed.

Starts drawing the shape at the location given by the integer values of numeric expressions *colh* and *rowh* . If you do not specify a location in the DRAW statement, the shape starts at the last point plotted by the last-executed DRAW, XDRAW, or HPLOT command.

The shape number specified (*exprnm*) must be between 0 and the number of shapes in the shape table (which must not exceed 255), inclusive.

Avoid using DRAW if there is no shape table in memory. The system might lock up, or you might find random shapes drawn on the screen. If your program extends into the high-resolution graphics portion of memory, some or all of it could be destroyed.

Not available in Integer BASIC.

## DSP

Displays the changing values of the specified variable as an Integer BASIC program progresses.

**Format:**

DSP *var*

The value of variable *var* and the current line number are displayed whenever the value of that variable changes. This display may interact with your program's output, rendering one or both illegible. RUN cancels all DSP instructions. Use CON or GOTO when you are debugging with DSP in immediate mode.

To turn off DSP, use NO DSP.

Not available in Applesoft.

**END**

Causes a program to halt.

**Format:**

END

No message is displayed. In Integer BASIC, END must be the last instruction executed or the warning \*\*\* NO END ERR is displayed. END is entirely optional in an Applesoft program.

Cannot be used in immediate mode in Integer BASIC.

**EXEC**

Executes a disk text file as if each character in the text file were entered on the keyboard.

**Format:**

EXEC *filename* [,R*n*] [,D*n*] [,S*n*] [,V*n*]

A text file to be used with EXEC consists of some combination of BASIC commands, DOS commands, and program lines. When EXEC is executed, the first line of the specified file is read from the disk. If a command, it is executed immediately; if a program line, it is added to any program lines in memory, just as if you had entered it directly from the keyboard.

An EXEC file can be used to enter an entire program, list it, run it, save it on the disk, change it, or anything else that can be done from the keyboard. You can even use an EXEC file to create and execute a second EXEC file.

The R parameter, if present, specifies which field of the file is first executed. When used with EXEC, the R parameter always counts from the beginning of the file: the first field in the file is field 0, the second is field 1, etc. The number following R must be an integer constant in the range 0 through 32767. If R*n* specifies the first field following the end of the file, nothing happens. If it specifies two fields or more past the end of the file, the END OF DATA message occurs.

If an INPUT statement is executed while an EXEC file is open, the response is taken from the EXEC file.

When the last line in the file has been executed, the EXEC file closes itself (see CLOSE). When an EXEC command is encountered in a controlling EXEC file, the original file is closed and any further commands in it are ignored; the new EXEC file is opened and executed normally.

If the file does not exist on drive  $Dn$  of slot  $Sn$ , the FILE NOT FOUND error message is displayed. If the disk in drive  $Dn$  of slot  $Sn$  is not volume  $Vn$ , the VOLUME MISMATCH error results.

$Dn$ ,  $Sn$ , and  $Vn$  can be specified in any order. If  $Dn$  or  $Sn$  is omitted, the last-referenced drive or slot is used.  $VO$  is used if  $Vn$  is absent. Also,  $n$  can be absent;  $DO$ ,  $SO$ , or  $VO$  will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## FLASH

This Applesoft statement turns on flashing video mode.

### Format:

FLASH

All output from subsequently executed PRINT statements will appear alternately as white-on-black and black-on-white characters. Error messages are similarly affected. However, characters echoed to the screen by INPUT statements are unaffected, as are any previously displayed characters.

FLASH works by slightly altering the standard ASCII codes. Therefore any characters sent to the disk in flash mode will be saved with incorrect codes. When read back in, the wrong characters will result.

Not available in Integer BASIC.

## FN

Listed in the Functions section of this chapter. See also DEF FN.

## FOR

Starts a loop that repeats a set of instructions until an automatically incremented variable attains a certain value.

### Format:

FOR  $varnm = exprm_1$  TO  $exprm_2$  [STEP  $exprm_3$ ]

When FOR is first executed  $varnm$  is assigned the value of  $exprm_1$ . The state-

ments following FOR are executed until a NEXT statement is reached. *varnm* is then incremented by *exprnm<sub>3</sub>* (or by 1 if the STEP clause is not present). After that, the new value of *varnm* is compared to the value of *exprnm<sub>2</sub>*. The sense of the comparison depends on the sign of *exprnm<sub>3</sub>*. If the sign is positive and the new value of *varnm* is less than or equal to *exprnm<sub>2</sub>*, execution loops back to the statement just after the FOR. The same thing happens if the sign of *exprnm<sub>3</sub>* is negative and the new value of *varnm* is greater than or equal to *exprnm<sub>2</sub>*. On the other hand, execution continues with the instruction that follows the NEXT if *varnm* is greater than *exprnm<sub>2</sub>* (*exprnm<sub>3</sub>* positive) or less than *exprnm<sub>2</sub>* (*exprnm<sub>3</sub>* negative). Because the comparison occurs *after* incrementing *varnm*, the instructions between FOR and NEXT are always executed at least once.

In Integer BASIC *varnm* must be an integer variable. In Applesoft *varnm* must be a real variable. It can never be a string variable.

The start, end, and increment values are determined from *exprnm<sub>1</sub>*, *exprnm<sub>2</sub>*, and *exprnm<sub>3</sub>* only once, on the first execution of the FOR statement. If you change these values inside the loop it will have no effect on the loop itself. You can change the value of *varnm* within the loop. This lets you terminate a FOR-NEXT loop before the end value is reached: set *varnm* to the end value (*exprnm<sub>2</sub>*), and on the next pass the loop will terminate itself. Do not start the loop outside a subroutine and terminate it inside the subroutine.

FOR-NEXT loops may be nested. Each nested loop must have a different *varnm* variable name. Each nested loop must be wholly contained within the next outer loop; at most, the loops can end at the same point. Integer BASIC allows 16 levels of FOR-NEXT nesting, Applesoft just 10.

FOR may be used in immediate mode only in Applesoft. The entire loop must be entered on one line. If NEXT is not present, the loop will execute once.

## FP

Places the Apple II in Applesoft.

### Format:

FP [*Dn*] [*Sn*] [*Vn*]

The source of Applesoft depends on what kind of Apple II you have, and what options are installed:

1. With an Apple II Plus the language is in read-only memory (ROM), no matter what options may also exist.
2. If you have the Applesoft firmware card installed, FP obtains the language from it regardless of the switch setting on the card.
3. With the Apple Language System installed, FP takes Applesoft from it.
4. On any other Apple II, FP looks for Applesoft on the specified (or current)

disk. If it does not exist there, the message LANGUAGE NOT AVAILABLE is displayed.

FP erases any BASIC program currently in memory.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. VO is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or VO will be used.

Use only in immediate mode.

## GET

This Applesoft statement accepts a single character from the keyboard without echoing it to the screen.

### Format:

GET *var*

Execution pauses until a key is depressed. When *var* is a string variable, the character entered is assigned to that variable. If CTRL-@ is entered, the null string is assigned to the variable.

GET is not often used with a numeric variable for *var*. When it is, entry of one of the digits 0 through 9 assigns that value to the variable. Entry of a plus sign, minus sign, comma, colon, CTRL-@, space, E, or period assigns a value of zero to the variable. Entering any character other than those just listed results in the message ?SYNTAX ERROR, and the program stops.

GET cannot be used in direct mode.

Not available in Integer BASIC.

## GOSUB

Causes the program to branch to the indicated line. When a RETURN statement is executed the program branches back to the instruction immediately following the GOSUB.

### General Format:

GOSUB *line*

The GOSUB statement calls a subroutine. The subroutine's entry point must occur on line number *line*. A subroutine's entry point is the beginning of the subroutine in a logical sense. That is to say, it is the line containing the statement

(or statements) which are executed first. The entry point need not necessarily be the subroutine line with the smallest line number.

Upon completing execution the subroutine branches back to the line following the GOSUB statement. The subroutine uses a RETURN statement in order to branch back in this fashion.

A GOSUB statement may occur anywhere in a program; in consequence a subroutine may be called from anywhere in the program.

Subroutines may be nested; that is to say, subroutines may be called from within subroutines. Twenty-five levels of nesting are allowed in Applesoft; that means 24 GOSUB statements may be executed before the first RETURN statement. The limit in Integer BASIC is 16 GOSUB statements.

Normally you must exit from a subroutine with a RETURN statement, not with a GOTO statement. But, you can use a GOTO statement to branch out of a subroutine if you first execute a POP statement.

Cannot be used in immediate mode in Integer BASIC.

#### **Additional Integer BASIC Format:**

GOSUB *exprnm*

In Integer BASIC a numeric expression is allowed in place of the line number. If *exprnm* does not evaluate to an existing line number, the message \*\*\* BAD BRANCH ERR is displayed. This form of GOSUB enables you to simulate the ON-GOSUB instruction, which is not available in Integer BASIC.

#### **GOTO**

Unconditionally causes program execution to branch to the line indicated.

#### **General Format:**

GOTO *line*

Program execution immediately continues with the first instruction in the line number indicated. If the line number does not exist the message ?UNDEF'D STATEMENT ERROR is displayed by Applesoft; the message \*\*\* BAD BRANCH ERR is displayed by Integer BASIC.

#### **Additional Integer BASIC Format:**

GOTO *exprnm*

In Integer BASIC a numeric expression is allowed in place of the line number. If *exprnm* does not evaluate to an existing line number the message \*\*\* BAD BRANCH ERR is displayed. This form of computed GOTO enables you to simulate the ON-GOTO statement, which is not available in Integer BASIC.



**GR**

Converts the screen to low-resolution graphics mode (40 × 40), leaving four lines for text at the bottom of the screen.

**Format** **GR**

The graphics portion of the screen is cleared to black, the cursor is moved to the text window, and COLOR is set to 0 (black).

If executed while HGR is in effect, GR behaves normally. However if HGR2 is in effect, you will be left looking at page 2 of low-resolution graphics and text. This can be confusing, as the screen will usually be filled with garbage and nothing you type will appear on the screen. To return to normal mode, type TEXT. Be sure to use TEXT in your programs before switching from HGR2 to GR.

You can switch to full-screen (40 × 48), low-resolution graphics with the statement POKE –16302,0 after executing GR. Anything you subsequently type in immediate mode will show up as color dots on the last four lines of the display screen, but will execute properly. POKE –16302,0 restores the text window.

**HCOLOR=**

This Applesoft statement sets the color for plotting in high-resolution graphics mode.

**Format:** **HCOLOR= *exprnm***

Until the next HCOLOR statement, all HPlot and DRAW statements will be executed in the color specified. The color codes are listed in Table 8-3. The value of *exprnm* must be in the range 0 through 7. Values outside this range will produce an ?ILLEGAL QUANTITY ERROR message. A high-resolution graphics plot executed before the first HCOLOR statement will be in an indeterminate color.

HCOLOR does not affect low-resolution graphics. An HCOLOR statement that

TABLE 8-3. High-Resolution Color Codes

Code	Color	Code	Color
0	Black	4	Black
1	Green	5	Orange*
2	Violet*	6	Blue*
3	White	7	White
* Depends on TV control settings			

is executed while the Apple II is not in high-resolution graphics mode does not affect the color of the next high-resolution graphics plot.

Not available in Integer BASIC.

## HGR

This Applesoft statement converts the screen to high-resolution graphics mode (280 × 160), with a four-line text window at the bottom.

**Format:** HGR

Page 1 of high-resolution screen memory is displayed. The low-resolution (text) screen memory is unaffected, but only the lowest four lines are visible. The cursor is not moved into this four-line text window. You might not be able to see it until you have typed several lines after executing HGR. The graphics portion of the screen is cleared to black. HCOLOR is left unchanged by this command.

You can switch to full-screen (280 × 192), high-resolution graphics with the statement `POKE -16302,0` after executing HGR. Any immediate mode commands you enter subsequently will not be visible but will still execute properly. `POKE -16301,0` restores the text window.

On Apple II systems with less than 32K bytes of memory, you cannot use HGR and the Disk Operating System (DOS) at the same time since they will try to use the same area of memory.

Furthermore, the Applesoft interpreter from disk or cassette occupies part of high-resolution graphics page 1 memory. Thus you cannot use HGR with disk-based or cassette-based Applesoft.

Even with firmware Applesoft, if your program is extremely long it might extend into high-resolution page 1. You can guard against this with the command `HIMEM: 8192` which keeps your program out of page 1.

Not available in Integer BASIC.

## HGR2

Converts the screen to full-screen, high-resolution graphics mode (280 × 192). Page 2 of high-resolution screen memory is displayed.

**Format:** HGR2

The low-resolution (text) screen memory is unaffected. Although you cannot see what you type, any command that you enter will be executed. The screen is cleared to black. HCOLOR is not affected by this command.

Page 2 of screen memory is not available if your system has less than 24K of memory. On 24K systems, set `HIMEM:` to 16384 before you use HGR2 to protect your program and variables from your graphics, and vice versa.

You cannot use HGR2 and the Disk Operating System concurrently unless your system has at least 36K bytes of memory. That means you need at least 36K to use HGR2 in disk-based Applesoft.

Do not try to establish a text window with POKE -16301,0. This will display low-resolution graphics page 2 while your immediate mode commands go into page 1 and hence are invisible (although they execute correctly).

Not available in Integer BASIC.

### HIMEM:

Sets an upper boundary on memory available to BASIC programs, including variable storage.

#### Format:

HIMEM: *exprnm*

HIMEM: establishes the highest location in read/write memory (RAM) available to your BASIC program. The Disk Operating System (DOS) always resides above HIMEM: if it is present. With the HIMEM: statement you can set aside additional space for machine language subroutines and high-resolution graphics shape tables. You can also protect the high-resolution graphics screen memory area of RAM.

HIMEM: is first set to the highest memory location on your Apple II (e.g., 49151 on a 48K Apple II). DOS resides in the highest part of memory, so it adjusts HIMEM: downward approximately 10,800 bytes when it is booted. Each additional file buffer you reserve via MAXFILES lowers HIMEM: another 595 bytes. If your Applesoft program uses strings, their values are stored starting at the resulting location of HIMEM:, working downwards. Refer to the memory map in Appendix G.

The value of *exprnm* must be in the range -65535 through 65535 (-32767 through 32767 in Integer BASIC) or an error message occurs.

You should not set HIMEM: higher than the maximum memory location available. If you do, some of your variable storage might end up in nonexistent memory.

You can see the current value of HIMEM: by using the appropriate instruction shown below:

```
PRINT PEEK(116) * 256 + PEEK(115)   for Applesoft
PRINT PEEK(77) * 256 + PEEK(76)     for Integer BASIC
```

HIMEM: is not affected by NEW, RUN, or CLEAR.

If you set HIMEM: lower than LOMEM:, or do not leave enough memory to run your program, an error message occurs.

Can only be used in immediate mode in Integer BASIC.

**HLIN**

Draws a horizontal line on the screen in low-resolution graphics mode.

**Format:**

**HLIN** *col<sub>1</sub>*, *col<sub>2</sub>* **AT** *row*

The line is drawn from *col<sub>1</sub>* to *col<sub>2</sub>* in the row specified. The color is determined by the **COLOR** statement last executed. If the screen is in text mode, or the text window is present and *row* is greater than 39, **HLIN** will draw a line of characters on the screen in the text window where the graphics dots would be plotted. The characters used are determined by previously executed **COLOR** statements; see Table 8-5 (near the **PLOT** statement in this chapter) for particulars.

In Integer BASIC, *col<sub>1</sub>* must be less than or equal to *col<sub>2</sub>* or the message **\*\*\* RANGE ERR** is displayed.

**HOME**

This Applesoft statement clears the display screen and positions the cursor to the upper lefthand corner (row 1 and column 1).

**Format:**

**HOME**

In Integer BASIC, use **CALL -936**.

**HPlot**

This Applesoft statement places a dot or draws a line of color on the high-resolution graphics screen.

**Formats:**

**HPlot** *colh*, *rowh*

**HPlot TO** *colh*, *rowh*

**HPlot** *colh<sub>1</sub>*, *rowh<sub>1</sub>* **TO** *col<sub>2</sub>*, *rowh<sub>2</sub>* [**TO** *colh<sub>3</sub>*, *rowh<sub>3</sub>*...]

The first form of the command places a dot of color on the screen at the specified location. The color of the dot is determined by the **HColor** statement last executed.

The second form of the command draws a line of color from the last dot plotted to the coordinates *colh* and *rowh*. If there has been no dot plotted since the last **HGR** or **HGR2** command, nothing will be plotted. The color of the line is deter-

mined by the HCOLOR statement last executed.

The third form of the command also draws a line of color. With firmware Applesoft, the line may have more than one segment. The line is first drawn from  $colh_1$  and  $rowh_1$  to  $colh_2$  and  $rowh_2$ . The color of the line (all segments) is determined by the HCOLOR statement last executed.

Additional coordinates are not allowed in cassette- or disk-based Applesoft. When present in a firmware Applesoft program, a new line segment is then drawn from  $colh_2$  and  $rowh_2$  to  $colh_3$  and  $rowh_3$ , and so on. There can be any number of coordinate pairs, as long as they all fit on one program line.

Any portion of a line or dot that lies within the text window will not be visible. However, if you switch to full-screen graphics with the command POKE -16302,0 any line or point plotted in the text window will then be visible.

You must always execute an HGR or HGR2 statement before an HPlot. Otherwise you may destroy your program or variables.

Not available in Integer BASIC.

## HTAB

This Applesoft statement positions the cursor to the specified column on the current display line.

### Format:

HTAB *col*

The cursor moves right or left to the column specified by the value of *col*, without erasing any displayed characters. Columns are numbered from 1 to 40 (left to right).

In Integer BASIC, use the TAB statement.

## IF-THEN

Conditionally causes the program to execute the indicated instruction or instructions. The rules for Integer BASIC and Applesoft are presented separately.

### Integer BASIC Formats:

IF *expr* THEN *statement*

IF *expr* THEN [GOTO] *line*

In the first form of the IF-THEN statement, the expression specifies a condition which, if true, causes the *statement* following the THEN to be executed. If the condition is false, the statement immediately following the IF-THEN *statement* is executed; the *statement* that follows THEN is not executed in this case.

In the second format of the IF-THEN statement (the conditional branch format),

the expression specifies a condition which, if true, causes the program to branch to the indicated line number.

Relational expressions are the most common type of expression used with IF-THEN. String values can only be compared for equality or nonequality in Integer BASIC.

*expr* can also be a numeric expression. In this case, *expr* is considered true if it has a nonzero value.

The IF-THEN expression cannot be a string expression (i.e., anything that evaluates to a string value) in Integer BASIC.

### Applesoft Formats:

```
IF expr THEN statement [:statement . . .]
IF expr { THEN
        GOTO
        THEN GOTO } line
```

In the first format of the IF-THEN statement, the expression specifies a condition which, if true, causes every *statement* that follows THEN on the same program line to be executed. If the specified condition is false, control passes to the first statement on the next program line and any *statements* following the THEN are not executed.

In the second format (the conditional branch format), the program branches to line number *line* if the condition is true. Otherwise execution continues with the first statement on the next program line after the IF-THEN.

If an unconditional branch is one of many statements following THEN, then the branch must be the last statement on the line, and it must have the GOTO *line* format. If the unconditional branch is not the last statement on the line, then statements following the unconditional branch can never be executed.

The most common type of expression used with IF-THEN is a relational expression. If string expressions are compared using relational operators, the ASCII codes (listed in Appendix I) for the characters involved determine the relative values of the strings. Strings are compared character by character until a mismatch occurs. Then the string with the higher ASCII code in the mismatch position is considered greater. If no mismatch occurs, the longer string is greater.

The expression may also be a numeric expression. If the value of the expression is not zero, the condition is considered true. If the value of the expression is zero (false), execution continues at the first statement on the next higher program line.

In Applesoft, *expr* may also be a string expression. However, execution of more than two or three of such IF-THEN statements in the course of a program generates the message ?FORMULA TOO COMPLEX ERROR.

Applesoft has problems if the last non-space character preceding THEN is the letter A. The A is combined with the T to form the reserved word AT. You can

avoid this problem by enclosing some or all of the expression (including the troublesome A) in parentheses.

If a FOR-NEXT loop follows the THEN, then the loop must be completely contained on the IF-THEN line. Additional IF-THEN statements may appear following the THEN as long as they are completely contained on the original IF-THEN line. However, a Boolean expression beats nested IF-THEN statements for clarity. For example, the two statements below are equivalent, but the second is easier to read.

```
10 IF A$ = "X" THEN IF B = 2 THEN IF C > D THEN 50
10 IF A$ = "X" AND B = 2 AND C > D THEN 50
```

### IN#

Selects the peripheral slot from which subsequent input will be accepted.

#### Format:

IN# *slot*

Subsequent INPUT statements will look for data from the peripheral in the slot indicated. *slot* must be an integer constant between 0 and 7. Note that slot 0 is not a peripheral device; IN# 0 specifies the keyboard as the input device. If there is no peripheral in the specified slot the system will lock up until you press RESET.

Whenever DOS is present in the Apple II memory, IN# is considered a DOS command, requiring PRINT and CTRL-D in programmed mode.

### INIT

Initializes a disk.

#### Format:

INIT *filename* [,D*n*] [,S*n*] [,V*n*]

The program currently in memory is saved on the disk under the file name given. This program becomes the greeting program, and is run automatically whenever this disk is booted. The disk is assigned the volume number it is initialized with; if no volume number is specified, the disk is assigned the default volume number of 254.

If the file does not exist on drive D*n* of slot S*n*, the FILE NOT FOUND error message is displayed.

D*n*, S*n*, and V*n* can be specified in any order. If D*n* or S*n* is omitted, the last-referenced drive or slot is used. Also, *n* can be absent; D0 or S0 will be used.

INIT may only be used in immediate mode.

## INPUT

Accepts character entry from the keyboard or other input device, evaluates it, and assigns the value or values entered to the variable or variables specified.

### Integer BASIC Format:

```
INPUT ["prompt",] var [,var . . .]
```

INPUT in Integer BASIC requests values for any combination of integer and string variables. If the first variable is an integer, then a question mark is displayed at the current cursor location as a cue to begin entry. Integer BASIC suppresses the question mark if a string is the first variable to receive input.

The optional *prompt* is a string constant. If it is present, it will be displayed just before the first variable is input; it is not repeated for each variable in the list. A question mark is displayed after the *prompt* if an integer variable is to be entered. The *prompt* alone is displayed if a string variable is to be entered. Note that the *prompt* is followed by a comma in the INPUT statement. The *prompt* may not be a string variable or string expression.

When a single INPUT statement calls for more than one integer value in succession, you can enter each one on a separate line; end each value with the RETURN key. Integer BASIC displays a double question mark (??) on each new line as a cue to continue entries for the INPUT statement. Optionally, you can enter more than one integer value on a single line; separate the values with commas.

Numeric input must consist only of valid numeric characters. These are the digits 0 through 9, spaces, and a plus or minus sign. You get an error message if you simply press RETURN when a numeric value is to be entered.

You must enter each string value on a separate line. All characters (except CTRL-C and CTRL-X) that you enter prior to pressing the RETURN key are accepted and assigned to the string variables. The null string ("" ) is assigned to the variable if you simply press RETURN when a string value is to be entered.

If you enter unacceptable characters (e.g., letters in a numeric value) the warning message \*\*\* SYNTAX ERR and RETYPE LINE appear. You must reenter all values that you entered on the offending line.

May not be used in immediate mode.

### Applesoft Format:

```
INPUT ["prompt";] var [,var . . .]
```

INPUT can request values for any combination of numeric and string variables. A question mark is normally displayed as a cue to begin entry at the current cursor location. Applesoft suppresses the question mark if the optional *prompt* is present.

The optional *prompt* is a string constant. If it is present, it will be displayed just before the first variable is input; it is not repeated for each variable in the list. No question mark is displayed after the *prompt*. Note that the *prompt* is followed by a



semicolon in the INPUT statement.

Generally speaking, when a single INPUT statement calls for more than one value, you can enter each one on a separate line; end each value with the RETURN key. Optionally, you can enter more than one value on a single line; separate the values with commas.

If you enter unacceptable characters (e.g., letters in a numeric value) a warning message appears and you must completely reenter the value. Applesoft displays REENTER and reexecutes the INPUT statement from the beginning. The cue (question mark or *prompt*) is redisplayed and you must reenter all values for the INPUT statement.

Numeric input must consist only of valid numeric characters. If you simply press RETURN when a numeric variable is to be entered, you receive an error message and must reenter the line. The digits 0 through 9, spaces, and a plus or minus sign are accepted as numeric input. Applesoft also accepts a decimal point, an additional plus or minus sign, and the letter E for entering real values and scientific notation.

In Applesoft, if the first nonspace character of a string entry is a quotation mark, all characters (including commas and colons) up to the next quotation mark or RETURN are assigned to the string variable. If the entry does not begin with a quotation mark, all characters (including quotation marks) up to the next comma, colon, or RETURN are assigned to the variable. If two or more strings are requested by the same INPUT statement, they must be enclosed in quotes and separated by commas.

If you simply press RETURN when a string variable is to be entered, the null string ("" ) is assigned to the variable.

In Applesoft, all characters after a colon in an INPUT response are ignored unless the entry begins with a quotation mark.

INPUT cannot be used in direct mode.

## INT

Places the Apple II in Integer BASIC.

**Format:**           INT

Any program currently in memory is erased. If Integer BASIC is not present (e.g., on an Apple II Plus without a Language System), the message LANGUAGE NOT AVAILABLE is displayed.

Use only in immediate mode.

## INVERSE

This Applesoft statement turns on inverse video mode (also called reverse video mode).

**Format:****INVERSE**

All output from subsequently executed PRINT statements will appear as black-on-white characters. Error messages are similarly affected. However, characters echoed to the screen by INPUT statements are unaffected, as are any previously displayed characters.

INVERSE works by slightly altering the standard ASCII codes. Therefore any characters sent to the disk in inverse mode will be saved with incorrect codes. When read back in, the wrong characters will result.

Not available in Integer BASIC.

**LET=**

The assignment statement, LET=, or simply =, assigns a value to a specified variable.

**Format:**

[LET] *var* = *expr*

Variable *var* is assigned the value computed by evaluating *expr*.

**LIST**

Displays all or part of the program currently in memory. There are two formats for the LIST command. One is recognized by both Integer BASIC and Applesoft, the other only by Applesoft. They are described separately below.

**General Format:**

LIST *line*<sub>1</sub> [, *line*<sub>2</sub>]

Any portion of the program may be listed. If no line numbers follow LIST, the entire program is displayed. If only *line*<sub>1</sub> is present, only that line will be displayed if it exists. If both line numbers are present, the program will be listed starting at *line*<sub>1</sub> and continuing through *line*<sub>2</sub>.

If *line*<sub>1</sub> does not exist, the listing starts at the next higher line number. If *line*<sub>2</sub> does not exist, the listing ends at the next lower line number.

LIST may not be used with variables or expressions in place of the line numbers.

When LIST displays your program it adds extra spaces around variables and reserved words to make the listing more readable. If this causes a problem, you can eliminate the spaces by reducing the text window to a width of 33 (or less) with the command POKE 33,33. (POKE 33,40 restores the text window to full width.)

Program line lengths are limited but these limits are calculated before LIST adds the extra spaces. You can therefore extend the apparent length of your program lines by leaving out spaces when you type the lines in. However, such a line will be too long to edit or copy after it has been listed with all the spaces put in.

#### Expanded (Applesoft) Format:

$$\text{LIST} \quad \left\{ \begin{array}{l} \text{line}_1 \text{ [ (,)]} \\ \text{[line}_1\text{] (,) line}_2 \end{array} \right.$$

In Applesoft, either a comma (,) or a hyphen (-) may separate the two line numbers.

In Applesoft you can list from the start of the program to a specific line number by putting a comma or hyphen ahead of *line*<sub>2</sub> (and omitting *line*<sub>1</sub>). You can also list from a specific line number to the end of the program by putting a comma or hyphen after *line*<sub>1</sub> (and omitting *line*<sub>2</sub>).

#### LOAD

Loads a program from cassette or disk.

#### Cassette Format:

LOAD

Loads the next sequential program from the cassette, replacing any program currently in memory. You must have the cassette recorder running in PLAY mode when LOAD is executed; the Apple II does not remind you to do this. The Apple II beeps as it starts to load a program and beeps again when it finishes. The second beep is your signal to manually stop the cassette recorder.

You can only use LOAD in immediate mode in Integer BASIC.

#### Disk Format:

LOAD *filename* [,D*n*] [,S*n*] [,V*n*]

The program with the name *filename* is loaded from the disk. If the LOAD is successful, any program previously in memory is erased.

If the program to be loaded is in Applesoft and the Apple II is currently in Integer BASIC, or vice versa, the Apple II switches to the proper language. This may require loading the language from the specified disk. If the language is not available, the message LANGUAGE NOT AVAILABLE is displayed.

If the file does not exist on drive D*n* of slot S*n*, the FILE NOT FOUND error message is displayed. If the disk in drive D*n* of slot S*n* is not volume V*n*, the VOLUME MISMATCH error results.

$Dn$ ,  $Sn$ , and  $Vn$  can be specified in any order. If  $Dn$  or  $Sn$  is omitted, the last-referenced drive or slot is used.  $VO$  is used if  $Vn$  is absent. Also,  $n$  can be absent;  $DO$ ,  $SO$ , or  $VO$  will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## LOCK

Protects a disk file from accidental erasure.

### Format:

LOCK *filename* [ $Dn$ ] [ $Sn$ ] [ $Vn$ ]

Once locked, a file cannot be deleted or renamed until it is unlocked (see UNLOCK). No program can be saved using the name of the locked file. A locked file is indicated in the disk catalog by an asterisk at the left of the file type.

If the file does not exist on drive  $Dn$  of slot  $Sn$ , the FILE NOT FOUND error message is displayed. If the disk in drive  $Dn$  of slot  $Sn$  is not volume  $Vn$ , the VOLUME MISMATCH error results.

$Dn$ ,  $Sn$ , and  $Vn$  can be specified in any order. If  $Dn$  or  $Sn$  is omitted, the last-referenced drive or slot is used.  $VO$  is used if  $Vn$  is absent. Also,  $n$  can be absent;  $DO$ ,  $SO$ , or  $VO$  will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## LOMEM:

Sets a lower boundary on memory available to BASIC programs for variable storage and the like.

### Format:

LOMEM: *exprnm*

LOMEM: establishes the lowest location in read/write memory (RAM) available for your BASIC program lines and variables. The Monitor and the BASIC interpreter use read/write memory below LOMEM: for pointers, low-resolution graphics and text screen memory, and so forth. When the Applesoft interpreter is loaded from disk or cassette it always resides in read/write memory below LOMEM:. You can set aside additional space for machine language subroutines and high-resolution graphics shape tables with the LOMEM: statement.

With Integer BASIC or firmware Applesoft, LOMEM: starts out at memory location 2048, just above the system-use area. Loading the Applesoft interpreter from disk or cassette raises LOMEM: to 12291. Each time you add an Applesoft program line or change an existing line, LOMEM: is adjusted up or down. Erasing an Applesoft program (with NEW or CTRL-B) also changes LOMEM:. So if you

want to reserve space below your program, you must do so after erasing any previous program but before loading or typing in the new one.

The value of *exprmm* must be in the range -65535 through 65535 (-32767 through 32767 in Integer BASIC), or an error message occurs.

You can display the current value of LOMEM: with the instruction PRINT PEEK(106) \* 256 + PEEK(105).

In Applesoft, if LOMEM: is set higher than the current value of HIMEM:, lower than the existing value of LOMEM:, or lower than the highest memory location used by the current operating system or program, then the message ?OUT OF MEMORY ERROR occurs.

Can only be used in immediate mode in Integer BASIC.

## MAN

Ends automatic line numbering mode in Integer BASIC.

### Format:

MAN

Automatic line numbering is instituted with AUTO.

Type CTRL-X to temporarily halt the generation of line numbers, then enter MAN.

Not available in Applesoft.

## MAXFILES

Specifies the maximum number of disk files that may be active at any one time.

### Format:

MAXFILES *limit*

The Disk Operating System (DOS) supports an absolute maximum of 16 open files at once. When executed, MAXFILES sets aside 595 bytes of memory (a file buffer) for each file. MAXFILES is automatically set to 3 when you boot DOS.

All DOS commands except MAXFILES use a file buffer while they are executing. Thus, the practical maximum number of files you may have open at any one time is one less than the MAXFILES limit. If you attempt to execute any DOS command when there is no buffer free, the error message NO BUFFERS AVAILABLE is generated.

MAXFILES resets HIMEM: when executed, which may erase part of your program, variable storage, etc. If possible, execute MAXFILES before you load or run your program.

If you use MAXFILES within an Applesoft program, use it as the first line. To

use MAXFILES in an Integer BASIC program, you must create an EXEC file as discussed in Chapter 5 (see also EXEC in this chapter).

*limit* must be an integer constant in the range 1 through 16 or the RANGE ERROR message occurs.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## MON

Causes disk commands and/or data flow to be displayed on the screen.

### Format:

MON [C] [,I] [,O]

The three parameters dictate what is displayed. If C is specified, all disk commands are displayed on the screen. If I is specified, all data input to the Apple II from the disk is displayed. If O is specified, all data output from the Apple II to the disk is displayed. These parameters may be used in any combination and in any order. If none of them is present, MON has no effect. MON remains in effect until a NOMON, FP, or INT is executed, the system is rebooted, or on some machines, RESET is struck.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## NEW

Deletes the current program and all variables from memory.

### Format:

NEW

NEW also resets LOMEM:, but does not affect HIMEM:, COLOR, or HCOLOR.

NEW may only be used in immediate mode in Integer BASIC.

## NEXT

Terminates the loop started by a FOR instruction.

### General Format:

NEXT *varnm* [, *varnm* . . .]

When NEXT is executed, loop index variable *varnm* is incremented by an amount specified in the corresponding FOR statement. The program then either continues with the instruction following NEXT or loops back to the corresponding FOR,

depending on the parameters set in the FOR statement. See the discussion of FOR earlier in this chapter.

If there is no currently active FOR loop that matches *varnm*, an error will occur. ?NEXT WITHOUT FOR ERROR is displayed by Applesoft; \*\*\* BAD NEXT ERR is displayed by Integer BASIC.

Multiple variables following NEXT must be listed in the proper order (the last loop initiated must be terminated first) or an error will occur.

NEXT may not be used in immediate mode in Integer BASIC. In Applesoft, an immediate mode NEXT may cause a branch to a FOR that was executed in programmed mode and is still active.

#### **Additional Applesoft Format:**

NEXT

In Applesoft you may use NEXT with no identifying variable name. The loop variable defaults to that of the most recently begun FOR loop that is still in effect. NEXT with no variable executes more rapidly than NEXT with a variable.

#### **NO DSP**

Cancels display mode for the specified variable in Integer BASIC. (See DSP.)

#### **Format:**

NO DSP *var*

Not available in Applesoft.

#### **NOMON**

Ends the display of disk commands or data flow that was initiated by MON.

#### **Format:**

NOMON [C] [,I] [,O]

Each parameter specified cancels part of the display started by MON. If C is specified, disk commands are not displayed. If I is specified, data input to the Apple II from the disk is not displayed. If O is specified, data output from the Apple II to the disk is not displayed. These parameters may be used in any combination and in any order. If MON is not in effect for the parameter or parameters specified, or no parameters are specified, NOMON has no effect.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

**NORMAL**

This Applesoft statement turns off FLASH and INVERSE video modes.

**Format:**

NORMAL

See FLASH and INVERSE.

Not available in Integer BASIC.

**NO TRACE**

Turns off the tracing of program execution that was initiated by TRACE.

**Format:**

NO TRACE

If TRACE is not in effect, NO TRACE has no effect.

**ONERR GOTO**

Branches to the line number indicated when a subsequent error occurs in an Applesoft program.

**Format:**

ONERR GOTO *line*

This command sets a flag that causes the program to branch to the line number indicated when an error occurs. ONERR GOTO must be executed before the error occurs.

Each type of error has a code number. The code of the most recently occurring error is stored in memory location 222. PEEK(222) retrieves the error codes. The error codes and their messages are listed in Table C-1 (Appendix C). When an error occurs inside a FOR-NEXT loop or in a subroutine the pointers and stacks are disrupted. Your error-handling routine must return to the FOR or GOSUB statement, restarting the loop or subroutine. If your error-handling routine returns to a NEXT or RETURN statement another error will occur.

ONERR GOTO will not work properly under some circumstances. The Apple II will lock up if there are two GET errors in a row and the error-handling routine ends with RESUME, not GOTO. In programs that use PRINT statements (or if TRACE is in effect), the 43rd error not arising from an INPUT statement causes a jump to the Monitor. In this situation, if GOTO ends the error-handling routine (instead of RESUME), the 87th INPUT error causes a jump to the Monitor.

You can circumvent all the problems just described if your error-handling



TABLE 8-4. Machine Language Fix for ONERR GOTO

MACHINE LANGUAGE		6502 ASSEMBLY LANGUAGE	
Decimal	Hexadecimal	Instruction	Comments
104	68	PLA	Put top byte of stack in Accumulator
168	A8	TAY	and save it in Y index register
104	68	PLA	Put next byte of stack in Accumulator
166	A6	LXD \$DF	Use ONERR pointer
223	DF		
154	9A	TXS	as stack address
72	48	PHA	Push saved stack contents on
152	98	TYA	'ONERR' stack (two bytes —
72	48	PHA	from Accumulator and Y register)
96	60	RTS	Return to Applesoft

routine includes a call to the machine language program in Table 8-4.

Use the POKE statement to put the decimal numbers into memory locations 768 through 777 (or any available memory locations). Then CALL 768 from your error-handling routine.

Not available in Integer BASIC.

Cannot be used in immediate mode.

## ON-GOSUB

Provides conditional subroutine calls to one of several subroutines in an Applesoft program, depending on the current value of an expression.

### Format:

**ON** *exprnm* **GOTO** *line [,line . . .]*

The program branches to the first line number if the integer value of the expression is 1, the second if it is 2, etc. The next RETURN statement encountered sends the program back to the statement following the ON-GOSUB.

The expression must have a value in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR occurs. If the expression evaluates to zero or to a value greater than the number of line numbers listed, program execution continues with the next instruction following the ON-GOSUB.

Not available in Integer BASIC. (But see GOSUB for an Integer BASIC form of computed GOSUB.)

## ON-GOTO

Causes a conditional branch to one of several points in an Applesoft program, depending on the current value of an expression.

### Format:

**ON *exprnm* GOTO *line* [,*line* . . .]**

The program branches to the first line number if the integer value of the expression is 1, the second if it is 2, etc.

The expression must have a numeric value in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR occurs. If the expression evaluates to zero or to a value greater than the number of line numbers listed, program execution continues with the next instruction following the ON-GOTO.

Not available in Integer BASIC. (But see GOTO for an Integer BASIC form of computed GOTO.)

## OPEN

Prepares a sequential or random-access disk text file for accessing.

### Format:

**OPEN *filename* [,*Ln*] [,*Dn*] [,*Sn*] [,*Vn*]**

A memory buffer of 595 bytes is allocated to the text file specified. The READ and WRITE commands can now be used with the file to retrieve and store information. If the file specified is not on the disk, one is created. If the file is already open, it is closed and then reopened.

If the L parameter is not specified, the file is opened as a sequential file.

If the L parameter is specified, the file is opened as a random-access file. The number *n* following L is the record length in bytes and must be an integer constant in the range 1 through 32767. It can be absent; L1 is used. Each time that a particular random-access file is opened, the record length must be the same.

If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

OPEN may not be used in immediate mode.

## PDL

Listed in the Functions section of this chapter.

PEEK

Listed in the Functions section of this chapter.

PLOT

Displays a point on the low-resolution graphics screen.

**Format:** `PLOT col, row`

In low-resolution graphics mode, PLOT places a dot of color on the screen. The color of the dot is determined by the COLOR statement last executed. Column numbers range between 0 and 39. Column 0 is at the left edge of the screen; column 39 is at the right. Row numbers range between 0 and 47. Row 0 is at the top of the screen; row 47 is at the bottom. A point plotted in rows 40 through 47 will be in the four-line text window unless a POKE -16302,0 has been executed to eliminate the text window.

In text mode or in the text window PLOT places a colored character, rather than a dot, on the screen. Since a character occupies the space of two vertically stacked graphics dots, there are two different sets of PLOT coordinates that will cause some character to appear in a given location. To place a particular character on the screen, you must PLOT both halves of the character location. The character that appears is determined by the COLOR statement last executed before each half is plotted.

Table 8-5 shows the characters generated by each combination of colors in the upper and lower graphics points. To generate a particular character, find it in Table 8-5; read up and left to find the colors that you must PLOT in each half of the character. The upper half of a character is specified by a PLOT in an even row, the lower half by a PLOT in an odd row. If you only plot one half of a character location, the character generated is based on the color of the PLOT and the color already present in the other graphics point.

TABLE 8-5. Characters Generated by Graphics Statements in Text Mode

	VIDEO MODE			COLOR OF UPPER GRAPHICS POINT															
	For Inverse Video	For Flashing Character	For Normal Character																
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
COLOR OF LOWER GRAPHICS POINT	0	4	8 12	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	1	5	9 13	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	2	6	10 14	b	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
	3	7	11 15	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

Note that each character can be generated by four different colors for the lower graphics point. They are not all identical, however. The character will be in reverse video if the color number of the lower point is less than 4, it will be flashing if the number is in the range 4 through 7, and it will be normal if the number is in the range 8 through 15.

## POKE

The POKE statement stores a byte of data in a specified memory location.

### Format:

POKE *memadr*, *byte*

A value between 0 and 255, provided by *byte*, is written into memory at location *memadr*. If the memory location specified exceeds the maximum location in memory (e.g., 16383 if you have 16K of memory), or accesses an output device that is not receiving, POKE has no effect.

Use caution with POKE. Some memory locations contain information essential to the Apple II's uninterrupted operation. Change random memory locations and you can destroy your program, lock up your system, or clobber your BASIC.

## POP

Causes the Apple II to forget the return location for the most recently executed GOSUB statement.

### Format:

POP

POP effectively changes the most recently executed GOSUB statement into a GOTO statement (ex post facto). The next RETURN statement executed will branch to the instruction immediately following the second most recently executed GOSUB. If the total number of POP and RETURN statements executed in a program exceeds the number of GOSUB statements executed, an error message occurs.

## POSITION

Moves the disk file pointer the specified number of fields ahead of its current position.

### Format:

POSITION *filename* [,R*n*]

If the file is not open when **POSITION** is executed, it is opened (see **OPEN**). The **R** parameter specifies how many fields the file pointer moves forward from its current position. The number following **R** must be an integer constant in the range 0 through 32767. If absent, this parameter defaults to zero, i.e., the file pointer is not moved. If the file is opened by **POSITION**, the fields are counted from the beginning of the file. The file must be a sequential file.

A field consists of a sequence of characters ending with a carriage return. **POSITION** goes through the file, character by character, counting such characters; the number of carriage return characters encountered is the number of fields skipped over. If any unused space is encountered in the file before the specified number of fields are counted, the message **END OF DATA** is given.

This is a DOS command, requiring **PRINT** and **CTRL-D** in programmed mode.

**POSITION** may not be used in immediate mode.

## **PR#**

Selects the peripheral slot that will receive subsequent output.

### **Format:**

**PR# slot**

Subsequent **PRINT** statements will send data to the peripheral in the slot indicated. *slot* must be an integer constant between 0 and 7. Note that slot 0 is not a peripheral device. **PR# 0** specifies the standard 40-column display screen as the output device. If there is no peripheral in the specified slot the system will lock up until you press **RESET**.

Whenever DOS is present in the Apple II memory, **PR#** is considered a DOS command, requiring **PRINT** and **CTRL-D** in programmed mode.

## **PRINT**

Outputs characters to the screen or other output device.

### **Format:**

**PRINT [expr] [{;}. . .[expr]]. . .]**

There are a number of acceptable variations on the **PRINT** statement. **PRINT** by itself outputs a carriage return and line feed. When **PRINT** is followed by one or more expressions, the values of these expressions are printed. The way the values appear depends on their nature and on the use of semicolons or commas in between values.

All numeric values in Integer BASIC and many in Applesoft are displayed using standard numeric representation. Negative values are preceded by a minus sign;

positive values are not preceded by a sign or a blank space. Scientific notation is used in Applesoft for values closer to zero than  $\pm .01$  and for any values with more than nine digits in front of the decimal point.

String values are displayed just as they are.

Commas and semicolons determine the spacing between printed values. A semicolon causes the next value to print immediately after the value just printed; they are concatenated with no intervening spaces. A comma causes the next value to print at the next tab stop, several spaces over from the last value.

In Integer BASIC, tab stops are eight characters apart, at columns 1, 9, 17, and so on. If any nonblank character is printed in the space just ahead of a tab stop (e.g., in column 16), that tab stop is inactivated.

Applesoft places tab stops 16 characters apart, at columns 1, 17, 33, and so on. Tab stops on the display screen will be inactivated according to a scheme illustrated in Figure 4-1 (Chapter 4). For other devices, a tab stop is inactivated if a nonblank character is printed just ahead of it (e.g., in column 32).

If the list of expressions does not end with a comma or semicolon, a carriage return and line feed are printed following the last item in the list. If the list ends with a semicolon the first character printed by the next PRINT statement will print directly following the last character printed by the current PRINT statement, with no intervening spaces. If the list ends with a comma the next output will print starting in the first position of the next tab field.

In Integer BASIC, all items must be separated by either a comma or a semicolon. In Applesoft, items may be listed with no intervening commas or semicolons. Output for such items is concatenated as if the items were separated by semicolons.

Applesoft recognizes a question mark (?) as an abbreviation for PRINT. The word PRINT will be spelled out when the program is listed, though.

## READ (DISK STATEMENT)

Specifies a disk file from which subsequent INPUT and GET commands will obtain data.

### Format:

**READ *filename* [,R*n*] [,B*n*]**

If the file specified is not already open, it is opened (see OPEN). All INPUT and GET statements receive characters from the disk until a disk statement (or a CTRL-D character (ASCII code 4) alone) is printed, RESET is struck, or an error occurs. If the file is not on the disk, the message FILE NOT FOUND appears.

The file will be read as a sequential file if the R parameter is absent. In a sequential file, the B parameter specifies at which byte (character) the READ begins. If there is no B parameter, READ begins at the first byte in the file (byte 0). If the byte to be read was never stored on the disk by a WRITE command, the END OF

DATA message is displayed when INPUT or GET is subsequently executed.

If the R parameter is present, the file will be accessed as a random-access file. The R parameter specifies which record of a random-access file will be READ.

In a random-access file, the B parameter specifies at which byte within the specified record the read begins. If there is no B parameter, READ begins at the first byte in the record (byte 0). If the byte to be read was never stored on the disk by a WRITE command, the END OF DATA message is displayed when INPUT or GET is subsequently executed.

The numbers following B and R must be integer constants in the range 0 through 32767. If unspecified, zero is assumed.

Do not use CTRL-C to stop a READ statement in Applesoft. This causes a series of ?REENTER messages to be displayed. Use only RESET to stop the program.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

May not be used in immediate mode.

## READ (ASSIGNMENT STATEMENT)

Assigns values from Applesoft DATA statements to variables.

### Format:

`READ var [, var . . .]`

There is a pointer to the DATA list which determines which value to assign to the first variable in the READ statement. At the start of the program and after a RESTORE statement, the pointer points to the first DATA value. As each READ statement variable gets a value, the pointer moves ahead to the next value.

The variables may be of any type, but must match the type of the corresponding DATA list values. A numeric value assigned to a string variable causes no problem. A string assigned to a numeric variable causes the message ?SYNTAX ERROR to be displayed. The line number of the offending DATA statement is announced with the error message.

If READ attempts to assign more variables than there are DATA values, the ?OUT OF DATA ERROR message appears, with the line number of the offending READ statement.

READ may be executed in immediate mode as long as the program in memory contains enough DATA values. Otherwise, the message ?OUT OF DATA ERROR occurs. If the Disk Operating System is present, a READ in immediate mode is interpreted as a DOS command, and the message NOT DIRECT COMMAND is displayed.

Not available in Integer BASIC.

## RECALL

Retrieves an Applesoft numeric array from cassette tape.

**Format:**

**RECALL *varnm***

Applesoft waits indefinitely until the array is found on the tape; no other instruction can be executed in the meantime. RECALL does not control tape movement nor does it advise when to start the cassette recorder in PLAY mode. There should be PRINT statements before and after RECALL which produce advisories. The Apple II does beep when it starts getting array values, and beeps again when it stops. The array must be dimensioned before the RECALL statement is executed, or the message ?OUT OF DATA ERROR is generated (see DIM).

You need not use the same array variable name in the RECALL statement as was used in the STORE statement for the same values. You should use an array with the same dimensions as the one that was stored, however. If the array that was stored contains more elements than the recalled array, the message ?OUT OF DATA ERROR occurs. If the recalled array contains at least as many elements as the stored array, but does not have exactly the same dimensions, the message ERR is generated, but program execution continues.

If the recalled array has more elements than the stored array, the values in the recalled array will usually be scrambled. There are two exceptions. You may recall into an array that has the same number of dimensions as the stored array, where each dimension except the last is the same size as the corresponding dimension in the stored array. The last dimension may be larger in the recalled array. You may also recall into an array with more dimensions than are in the stored array, if the dimensions that are in the array match the corresponding dimensions in the recalled array (or exceed them, in the case of the last dimension of the stored array).

String arrays cannot be used with RECALL. But recalled numeric values can be converted to string values with the CHR\$ function.

Not available in Integer BASIC.

**REM**

The Remark statement (REM) allows comments to be placed in the program for program documentation purposes.

**Format:**

**REM *comment***

*comment* is any sequence of characters that will fit on the current program line.

Remark statements are reproduced in program listings, but they are otherwise ignored. A REM statement may be placed on a line of its own or it may be placed as the last statement of a multiple-statement line.

REM cannot be placed ahead of any other statements on a multiple-statement line, since all text following the REM is treated as a comment.



## RENAME

Changes the name of a disk file without altering the file contents.

### Format:

RENAME *filename<sub>1</sub>*, *filename<sub>2</sub>* [,D*n*] [,S*n*] [,V*n*]

The file with the name *filename<sub>1</sub>* is found on the disk, and its name is changed to *filename<sub>2</sub>*. If the file is open, it is closed (see CLOSE). The file is not affected in any other way.

RENAME will readily change the file name to one that already exists on the disk; in fact it will do this any number of times. You must make sure that there is no file already named *filename<sub>2</sub>* before RENAME is executed.

If *filename<sub>1</sub>* does not exist on drive D*n* of slot S*n*, the FILE NOT FOUND error message is displayed. If the disk in drive D*n* of slot S*n* is not volume V*n*, the VOLUME MISMATCH error results.

D*n*, S*n*, and V*n* can be specified in any order. If D*n* or S*n* is omitted, the last-referenced drive or slot is used. V0 is used if V*n* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## RESTORE

Resets the Applesoft DATA list pointer to the beginning of the list.

### Format:

RESTORE

Subsequent READ statements start at the first DATA value.

Not available in Integer BASIC.

## RESUME

Causes an Applesoft program to resume execution at the beginning of the instruction in which an error occurred.

### Format:

RESUME

RESUME may only be used after an ONERR GOTO branch has been triggered by an error. If RESUME is executed when no error has occurred, the results are unpredictable but generally tragic.

Not available in Integer BASIC.

Cannot be used in immediate mode.

## RETURN

Causes the program to branch to the statement immediately following the most recently executed GOSUB.

**Format:**                      RETURN

The POP statement will obliterate all knowledge of the most recent GOSUB, so RETURN after POP causes a branch to the statement following the next most recent GOSUB.

If more RETURN (and POP) than GOSUB statements are executed in a program, an error message occurs.

## ROT=

This Applesoft statement sets the orientation of high-resolution shapes drawn by DRAW or XDRAW.

**Format:**                      ROT=*exprnm*

ROT=0 draws the shape in the orientation with which it was defined. The shape is rotated 90 degrees clockwise for each increment of 16 in the value of *exprnm*. Thus, ROT=32 draws the shape upside down, and ROT=64 draws the shape in its original orientation. Values for *exprnm* greater than 64 are evaluated modulo 64.

When SCALE has been set at 1, there are only four recognized values for ROT=. They are 0, 16, 32, and 48 (and values greater than 63 equivalent to these values). When SCALE=2 there are eight values, when SCALE=3 there are 16 values, etc., up to a maximum of 64 different recognized values. An unrecognized value for ROT= will be treated as if it were the next lower recognized value.

The *exprnm* must have a value in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR is generated when the ROT= command is executed.

ROT= is not recognized as a reserved word unless the character "=" is the first following nonspace character.

Not available in Integer BASIC.

## RUN (DISK STATEMENT)

Loads and runs a program from a disk.

**Format:**                      RUN *filename* [,D*n*] [,S*n*] [,V*n*]

The program named *filename* is loaded from the disk and then run. If the load is successful, any program previously in memory is erased.

If the program to be loaded and run is in Integer BASIC and the Apple II is currently in Applesoft, or vice versa, the Apple II switches to the proper language. If necessary, it will load the Applesoft interpreter from the specified disk. If the language is not available, the message LANGUAGE NOT AVAILABLE is displayed.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

### **RUN (GENERAL STATEMENT)**

Executes the program currently in memory, starting at the specified line number, if present; otherwise at the lowest numbered line in the program.

#### **General Format:**

RUN [*line*]

If there is no such line number as *line* you will receive the \*\*\* BAD BRANCH ERR from Integer BASIC or the ?UNDEF'D STATEMENT ERROR from Applesoft.

#### **Additional Integer BASIC Format:**

RUN *exprnm*

In Integer BASIC, the line number can be a numeric expression.

RUN may only be used in immediate mode in Integer BASIC.

### **SAVE**

Saves the program currently in memory on cassette or disk.

#### **Cassette Format:**

SAVE

Saves the program currently in memory on cassette tape. You must have the cassette recorder running in RECORD mode when SAVE is executed. The Apple II does not remind you to do this. The Apple II beeps as it starts to save a program and beeps again when it is finished. The second beep is your signal to manually stop the cassette recorder.

May only be used in immediate mode in Integer BASIC.

**Disk Format:**

**SAVE *filename* [,D*n*] [,S*n*] [,V*n*]**

If there is no file with the name *filename* on the disk, a file is created with that name and in the language of the current program. The program is saved. If there is a file named *filename* in the same language as the current program, the contents of that file are erased and the current program is saved in their place. If program *filename* exists but in a different language or with a different file type, the message FILE TYPE MISMATCH occurs.

If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

**SCALE**

This Applesoft statement sets the size of high-resolution graphics shapes drawn by DRAW or XDRAW.

**Format:**

**SCALE= *exprnm***

The size of the shape in the shape table is multiplied by the integer value of *exprnm*. Thus, if SCALE=1 the shape will be drawn just as it was defined, if SCALE=2 it will be drawn twice that size, etc. If SCALE=0 the shape is drawn 255 times the size of the original.

The value of *exprnm* must be in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR occurs when the SCALE command is executed.

SCALE is not recognized as a reserved word unless the character "=" is the first following nonspace character.

Not available in Integer BASIC.

**SHLOAD**

This Applesoft statement loads a high-resolution graphics shape table from cassette tape.

**Format:**

**SHLOAD**

Constructing and storing a shape table on cassette tape is discussed in Chapter 6.

The shape table is loaded into memory just below HIMEM: and HIMEM: is set just below the shape table. The starting location of the table is stored in memory locations 22 and 23.

Before you execute SHLOAD be sure that you have set HIMEM: so that the shape table will not be loaded on top of your program or variables and will not be erased by your graphics. Refer to the discussion of HIMEM: in this chapter, memory maps in Appendix G, and to Chapter 6 for more information.

Not available in Integer BASIC.

## SPEED

This Applesoft statement changes the rate at which characters are output.

### Format:

SPEED *exprnm*

The value of *exprnm* establishes the rate at which characters appear on the display screen or other output device. Speeds range from 0 (slowest) to 255 (fastest).

Not available in Integer BASIC.

## STOP

Causes an Applesoft program to halt execution.

### Format:

STOP

The Apple II returns to immediate mode. The message BREAK IN *line* is displayed, where *line* is the line number at which the STOP was executed.

Not available in Integer BASIC.

## STORE

Saves the specified Applesoft array on cassette tape.

### Format:

STORE *varnm*

STORE does not control tape movement nor does it advise when to start the cassette recorder in RECORD mode. You must have the cassette recorder running and ready to record when STORE is executed. Your Applesoft program should display advisories (via PRINT statements). The Apple II does beep when it starts

saving values and beeps again when it stops.

You may only STORE numeric arrays; string arrays must be converted to integer values using the ASC function in order to be stored (see also RECALL).

Not available in Integer BASIC.

## TAB

This Integer BASIC statement positions the cursor to the specified column on the current display line.

### Format:

TAB *col*

The cursor moves right or left to the column specified by the value of *col*, without erasing any displayed characters. Columns are numbered from 1 to 40 (left to right).

For Applesoft, use the HTAB statement. See also the TAB function listed in the Functions section of this chapter.

## TEXT

Returns the screen to the usual full-screen text mode from any of the graphics modes.

### Format:

TEXT

The prompt and cursor are moved to the last line of the screen; if issued in text mode, this is the only result. If the text window has been set to anything other than full-screen, TEXT resets to full-screen.

TEXT does not clear the screen, or more precisely, does not clear page 1 of low-resolution screen memory. Since the normal text mode uses the same screen memory as low-resolution graphics, executing TEXT while in low-resolution graphics mode will leave the top 20 lines of the screen filled with strange characters.

## TRACE

Displays the line number of each statement as it is executed.

### Format:

TRACE

This debugging aid may cause line numbers to display intermixed with your program's output, rendering one or both illegible. TRACE is only turned off by NO TRACE.

## UNLOCK

Removes locked status from a disk file, permitting it to be changed or deleted.

### Format:

UNLOCK *filename* [,*Dn*] [,*Sn*] [,*Vn*]

If the file specified is locked, the lock is removed. If the specified file is not locked, nothing happens (see LOCK).

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## USR

Listed in the Functions section of this chapter.

## VERIFY

Checks a specified disk file for self-consistency.

### Format:

VERIFY *filename* [,*Dn*] [,*Sn*] [,*Vn*]

When a file is saved on a disk with a SAVE, BSAVE, or PRINT statement, a checksum is calculated for each sector and stored on the disk. VERIFY recalculates these checksums and compares them to the checksums on the disk. If they match, the file is intact and no message is returned. If one or more do not match, the message I/O ERROR is generated. Any type of file may be verified.

If the file does not exist on drive *Dn* of slot *Sn*, the FILE NOT FOUND error message is displayed. If the disk in drive *Dn* of slot *Sn* is not volume *Vn*, the VOLUME MISMATCH error results.

*Dn*, *Sn*, and *Vn* can be specified in any order. If *Dn* or *Sn* is omitted, the last-referenced drive or slot is used. V0 is used if *Vn* is absent. Also, *n* can be absent; D0, S0, or V0 will be used.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

## VLIN

Draws a vertical line on the screen in low-resolution graphics.

### Format:

**VLIN** *row<sub>1</sub>*, *row<sub>2</sub>* AT *col*

The line is drawn from *row<sub>1</sub>* to *row<sub>2</sub>* in the column specified by *col*. The color is determined by the COLOR statement last executed. If the screen is in text mode, or the text window is present and either row is greater than 39, some or all of the line will appear as characters instead of graphics dots. The characters used are determined by previously executed COLOR statements; see Table 8-5 (near the PLOT statement in this chapter) for particulars.

In Integer BASIC, *row<sub>1</sub>* must be less than or equal to *row<sub>2</sub>* or the message \*\*\* RANGE ERR will be displayed.

## VTAB

Positions the cursor to the specified line in the current display column.

### Format:

**VLIN** *row*

The cursor moves up or down to the line specified by the value of *row*, without erasing any displayed characters. Rows are numbered from 1 to 24 (top to bottom).

## WAIT

Halts an Applesoft program until a particular memory location attains a specified condition.

### Format:

**WAIT** *memadr*, *exprnm*, [, *exprnm<sub>2</sub>*]

WAIT checks all or part of the eight bits of memory location *memadr* for the pattern of ones and zeros specified by the binary value of *exprnm<sub>2</sub>*. The binary value of *exprnm<sub>1</sub>* determines which bits of the memory location to consider and which to ignore. If a particular bit of *exprnm<sub>1</sub>* is 1, then the corresponding bit of memory location *memadr* is checked. Conversely, WAIT ignores those memory bits that



correspond to 0 bits in the binary value of *exprnm<sub>2</sub>*.

As long as the significant bits (as determined by *exprnm<sub>1</sub>*) of *memadr* are all different from the corresponding bits of *exprnm<sub>2</sub>*, the wait continues. The moment any pair of significant bits are the same (either both 0 or both 1) the wait is over and the Applesoft program continues.

If *exprnm<sub>2</sub>* is absent, 0 is used.

WAIT can only be interrupted by RESET (or power off). The value of the numeric expressions must be in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR is generated. If the specified memory location is greater than the maximum location in memory (e.g., 32767 if you have 32K of memory), or accesses an output device that is not receiving, WAIT will lock up the system until you press RESET.

Not available in Integer BASIC.

## WRITE

Specifies a disk file to which subsequent PRINT statements will send output.

### Format:

WRITE *filename* [,R*n*] [,B*n*]

If the file specified is not already open, it is opened (see OPEN). Subsequent PRINT statements save data on the disk until a disk statement (or a CTRL-D character (ASCII code 4) alone) is printed. If the file is not on the disk, the message FILE NOT FOUND appears.

The file will be written to as a sequential file if the R parameter is absent. In a sequential file, the B parameter specifies at which byte (character) the WRITE begins. If there is no B parameter, WRITE begins at the first byte in the file (byte 0).

If the R parameter is present, the file will be written to as a random-access file. The WRITE is to the record specified by the R parameter.

In a random-access file, the B parameter specifies at which byte within the specified record the WRITE begins. If there is no B parameter, WRITE begins at the first byte in the record (byte 0).

The B parameter can be used to write starting at a point beyond the last character already in the file (or record). This data can be READ, but any attempt to READ intervening unused bytes generates the OUT OF DATA message.

The numbers following R and B must be integers in the range 0 through 32767. If unspecified, 0 is assumed.

While WRITE is in effect, every character that the Apple II outputs that would normally be sent to the screen is sent to the disk. This includes the question mark generated by INPUT and any error messages.

This is a DOS command, requiring PRINT and CTRL-D in programmed mode.

May not be used in immediate mode.

## XDRAW

This Applesoft statement draws a high-resolution graphics shape on the screen and, if used a second time with the same parameters, erases that shape.

### Format:

**XDRAW *exprnm* [AT *colh*, *rowh*]**

Shape number *exprnm* from the shape table is drawn, with each point in the color that is the complement of the color on the screen at that point. Colors 0 and 3 are a complementary pair, as are 1 and 2, 4 and 7, and 5 and 6 (see Table 8-3). The scale and rotation of the shape must be set by SCALE and ROT commands before the XDRAW command is executed.

You use XDRAW instead of DRAW so that you can easily erase a shape you have drawn. Since XDRAW draws in the color complementary to the color previously at that point, if you execute two (or four, six, etc.) XDRAW statements with the same parameters, whatever is on the screen will be unchanged.

If you do not specify a location in the XDRAW statement, the shape is drawn starting at the point plotted by the last executed DRAW, XDRAW, or HPLOT command. If you do specify a location, the shape is drawn starting at that point (*colh*, *rowh*).

The shape number, *exprnm*, must have a value between 0 and the number of shapes in the shape table (which must not exceed 255), inclusive.

Not available in Integer BASIC.

## FUNCTIONS

Apple II BASIC functions are described below in alphabetical order. Nomenclature and abbreviations are described at the beginning of this chapter.

Many of the functions are available only in Applesoft. Such functions are appropriately identified.

### ABS

Returns the absolute value of a number. This is the value of the number without regard to sign.

### Format:

**ABS *exprnm***

**ASC**

Returns the ASCII code number for a specified character.

**Format:**

**ASC (*expr*\$)**

If the string is longer than one character, ASC returns the ASCII code for the first character in the string. The code returned will not necessarily be the lowest ASCII code (in the range 0-95) for that character. The characters generated by ASCII codes between 96 and 255 duplicate those in the lower range on the display screen. However, they are not evaluated as the same character by relational operators such as  $<$ ,  $>$ , and  $=$ . They may be treated differently by printers and other output devices as well. If the first character of *expr*\$ is CTRL-@ (ASCII code 0), the message ?SYNTAX ERROR is generated. If *expr*\$ is a null string the message ?ILLEGAL QUANTITY ERROR is produced.

ASCII codes are listed in Appendix I.

**ATN**

Returns the arctangent of the argument.

**Format:**

**ATN (*exprnm*)**

Computes the arctangent, in radians, of *exprnm*. The angle returned is in the range  $-\pi/2$  through  $\pi/2$ .

Not available in Integer BASIC.

**CHR\$**

Returns the string value of the specified ASCII code.

**Format:**

**CHR\$ (*exprnm*)**

Returns the character represented by the integer value of *exprnm*, interpreted as an ASCII code. You will find a table of ASCII character codes in Appendix I. Use this function to generate characters you cannot produce at the keyboard for controlling peripheral devices, etc. The value of *exprnm* must be in the range 0 through 255 or the message ?ILLEGAL QUANTITY ERROR will appear.

Not available in Integer BASIC.

**COS**

Returns the cosine of an angle.

**Format:** **COS (*exprnm*)**

Computes the cosine of *exprnm* radians.

Not available in Integer BASIC.

**EXP**

Returns *e* raised to a power.

**Format:** **EXP (*exprnm*)**

Computes *e* (the base of natural logarithms, 2.71828183) raised to the power *exprnm*.

Not available in Integer BASIC.

**FN**

Invokes a previously executed user-defined function.

**Format:** **FN *varnm* (*exprnm*)**

*varnm* is the name of the function. The value of the *exprnm* is assigned everywhere the dummy variable occurs in the function definition, and the resulting expression is evaluated. See DEF FN in the Statements section of this chapter.

A function may not be recursive, i.e., *exprnm* may not refer to FN *varnm* nor to any other function which refers to FN *varnm*.

If you attempt to use FN *varnm* before the DEF FN *varnm* statement has been executed you will receive the ?UNDEF'D FUNCTION ERROR message.

Not available in Integer BASIC.

**FRE**

Returns the number of bytes of memory currently available to an Applesoft program.

**Format:** **FRE (*exprnm*)**

The memory available to you is that below the string storage area and above the array storage. If there are more than 32767 bytes of memory available, FRE returns a negative number. Add 65536 to this number to discover the actual amount of memory available.

FRE also clears disused strings from the string storage area. When a string changes value during a program the old value of the string is left in memory, and the new value is added to the string storage area. Eventually this might infringe on memory you are using for something else. To prevent this problem, have a statement such as `A = FRE (0)` executed periodically in programs that use strings extensively.

The value of *exprnm* is not used by FRE, but it will cause an error if it is illegal.  
Not available in Integer BASIC.

## INT

Returns the integer portion of a number.

### Format:

INT (*exprnm*)

Returns the largest integer less than or equal to the value of *exprnm*.

Not available in Integer BASIC.

## LEFT\$

Returns the leftmost characters of a string.

### Format:

LEFT\$ (*expr\$*, *exprnm*)

Returns the leftmost *exprnm* characters of *expr\$*. *exprnm* must be in the range 1 through 255, and *expr\$* may not have more than 255 characters. If *exprnm* is greater than the length of *expr\$*, the entire string is returned.

Not available in Integer BASIC.

## LEN

Returns the length of a string.

### Format:

LEN (*expr\$*)

Counts the number of characters in *expr\$*, including all spaces and nonprinting

characters. If *expr\$* has more than 255 characters (possible only if *expr\$* is a string expression involving concatenation) the message ?STRING TOO LONG ERROR is generated.

## LOG

Returns the natural logarithm of a number.

### Format:

LOG (*exprnm*)

Computes the natural logarithm of *exprnm*. Returns ?ILLEGAL QUANTITY ERROR if *exprnm* is zero or negative.

Not available in Integer BASIC.

## MID\$

Returns any specified portion of a string.

### Format:

MID\$ (*expr\$*, *exprnm*<sub>1</sub> [, *exprnm*<sub>2</sub>])

Returns *exprnm*<sub>2</sub> characters from *expr\$*, starting with the character *exprnm*<sub>1</sub>. If *exprnm*<sub>2</sub> is absent, MID\$ returns the portion of *expr\$* from the character *exprnm*<sub>1</sub> through the last character. If the length of *expr\$* is less than *exprnm*<sub>1</sub>, the null string is returned. If there are fewer than *exprnm*<sub>2</sub> characters in *expr\$* after *exprnm*<sub>1</sub>, the result is the same as if *exprnm*<sub>2</sub> were absent. *expr\$* must not exceed 255 characters, and *exprnm*<sub>1</sub> and *exprnm*<sub>2</sub> must each be in the range 1 through 255.

Not available in Integer BASIC.

## PDL

Returns the current value of the game control (paddle) specified.

### Format:

PDL (*exprnm*)

The value returned is an integer between 0 and 255 based on the rotation of paddle number *exprnm*, or the resistance of a device connected to game control socket *exprnm*. The game controls are numbered 0 through 3. If the paddle number is less than 0 or greater than 255 the message ?ILLEGAL QUANTITY ERROR is displayed. If the paddle number is between 4 and 255, PDL returns a somewhat

unpredictable number between 0 and 255, and may cause various side effects such as a click from the speaker or a sudden shift in graphics mode.

If two PDL instructions are executed consecutively or nearly consecutively, the second value may be affected by the first. Make sure that several instructions are executed between PDL functions (an empty FOR-NEXT loop will do).

## PEEK

Returns the contents of a memory location.

### Format:

PEEK (*memadr*)

The value returned is the decimal equivalent of the eight bits at memory location *memadr*. Appendix E lists some useful memory locations.

## POS

Returns the column position of the cursor.

### Format:

POS (*exprnm*)

The expression is a dummy; it is not used and therefore can have any legal value.

POS will return a value between 0 and 39. Character positions begin at 0 for the leftmost character.

Not available in Integer BASIC.

## RIGHT\$

Returns the rightmost characters of a string.

### Format:

RIGHT\$ (*expr\$*, *exprnm*)

Returns the rightmost *exprnm* characters of *expr\$*. The value of *exprnm* must be in the range 1 through 255, and *expr\$* may not have more than 255 characters. If *exprnm* is greater than the length of *expr\$*, the entire string is returned.

Not available in Integer BASIC.

## RND

Returns a random number.

**Format:****RND (*exprnm*)**

Returns a random number, the range of which depends on the value of *exprnm* and the version of BASIC.

In Integer BASIC, RND returns a random integer between 0 and the value of *exprnm*, exclusive of *exprnm* but inclusive of 0. Thus, RND (1) always returns 0, and RND (-2) produces a fifty-fifty mix of 0 and -1. Attempting to use RND (0) causes the message \*\*\*>32767 ERR to be displayed.

In Applesoft, RND always returns a real number greater than or equal to 0 and less than 1. The value returned can be one of three types, depending on the sign of *exprnm*.

If *exprnm* is positive, RND returns a different value each time it is used, unless a repeatable sequence has been started.

A repeatable sequence starts when RND is used with a negative *exprnm*. Any particular negative value always starts the same sequence; subsequent positive arguments will return a repeatable sequence of random numbers. A different repeatable sequence is started by each different negative value of *exprnm*. This feature is useful for testing and debugging programs that use RND.

If *exprnm* is 0 in Applesoft, RND returns the random number most recently generated (this is not affected by CLEAR or NEW).

**SCRN**

Returns the color code of the low-resolution graphics point with the specified coordinates.

**Format:****SCRN (*col*, *row*)**

If the screen is in text mode, or the text window is present and the point specified is within it, SCRn returns the color code of half of the character. The color code of the top half of the character is returned if *row* is even, that of the bottom half if *row* is odd. The ASCII code of the character at character position (*a*, *b*) (with *a* in the range 0-39 and *b* in the range 0-23) is returned by the expression  $SCRn(a, 2*b) + 16*SCRn(a, 2*b + 1)$ . Thus the character itself is returned by  $CHR$(n)$ , where *n* is the value returned by the above expression.

If *col* is in the range 0-39, SCRn returns the color code of the graphics point (*col*, *row*). If *col* is in the range 40-47 and *row* is in the range 0-31, SCRn returns the color number of the graphics point (*col*-40, *row*+16). If *col* is in the range 40-47 and *row* is in the range 32-47, SCRn returns a number unrelated to anything on the screen.

If SCRn is used while the screen is in high-resolution graphics mode, the number returned is related to the low-resolution graphics area of memory rather than the high-resolution display.



SCRN is only recognized as a reserved word if the next nonspace character is a left parenthesis.

## SGN

Determines whether a number is positive, negative, or zero.

### Format:

SGN (*exprnm*)

The SGN function returns +1 if *exprnm* is positive, -1 if it is negative, and 0 if it is zero.

## SIN

Returns the sine of an angle.

### Format:

SIN (*exprnm*)

Computes the sine of *exprnm* radians.

Not available in Integer BASIC.

## SPC

Moves the cursor right a specified number of positions.

### Format:

SPC (*exprnm*)

The SPC function is used in PRINT statements to print *exprnm* blank spaces. Therefore any characters which the cursor passes over are erased.

The SPC function moves the cursor rightward from whatever column position the cursor happens to be at when the SPC function is encountered. This is in contrast to a TAB function, which moves the cursor to some fixed column measured from the leftmost column of the display.

Not available in Integer BASIC.

## SQR

Returns the square root of a positive number.

**Format:**  $\text{SQR}(\text{exprnm})$

A negative value of *exprnm* causes the ?ILLEGAL QUANTITY ERROR message.  $\text{SQR}(\text{exprnm})$  operates faster than  $(\text{exprnm})^{(.5)}$ .

Not available in Integer BASIC.

## STR\$

Converts a numeric value to a string.

**Format:**  $\text{STR\$}(\text{exprnm})$

The value of *exprnm* is converted to a string. The string characters are the same as those that would be printed by a `PRINT exprnm` statement. Therefore,  $\text{STR\$}(2/3) = ".666666667"$  and  $\text{STR\$}(2468013579) = "2.46801358E+09"$ . If *exprnm* exceeds the limits for real numbers, the message ?OVERFLOW ERROR is displayed.

Not available in Integer BASIC.

## TAB

TAB moves the cursor right to the specified column position.

**Format:**  $\text{TAB}(\text{exprnm})$

Use TAB with the PRINT statement to move the cursor to column *exprnm*, if *exprnm* is to the right of the cursor's current position. The cursor does not move if *exprnm* is not to the right of the current position. TAB prints blank spaces as it moves the cursor right, thereby erasing anything that was on the screen beforehand.

For TAB, columns are numbered from 1 to 255. If *exprnm* is larger than the width of the output device (40 for the display screen), it moves the cursor down one line and resumes counting at the left margin. If the value of *exprnm* is 0, TAB moves to column 256. A value of *exprnm* outside the range 0 to 255 causes the error message ?ILLEGAL QUANTITY ERROR.

See also HTAB (Applesoft) and TAB (Integer BASIC) in the Statements section of this chapter.

Not available in Integer BASIC.

## TAN

Returns the tangent of an angle.

**Format:**

TAN (*exprnm*)

Computes the tangent of *exprnm* radians.

Not available in Integer BASIC.

**USR**

Branches to a machine language subroutine, passing values in the Accumulator.

**Format:**

USR *exprnm*

The subroutine starts at memory location 10 (0A hexadecimal). Locations 10 through 12 (0A through 0C hexadecimal) must contain an assembly language JMP instruction that branches to the starting location of your subroutine. Since USR is a function, it returns a numeric real value. Whatever is in the Accumulator when the assembly language subroutine executes an RTS instruction (returning to the Applesoft program) is the value returned.

There are many useful machine language subroutines present in the Apple II Monitor. They are listed in Appendix D.

See also the CALL statement described in the Statements section of this chapter, which is available in Integer BASIC also.

Not available in Integer BASIC.

**VAL**

VAL converts a string to a numeric value.

**Format:**

VAL (*expr\$*)

Returns the numeric value represented by *expr\$*. If the first character of *expr\$* is not a numeric character, zero is returned. Otherwise, *expr\$* is taken character by character until an unacceptable character is encountered. The acceptable characters are: the digits 0 through 9, spaces, a decimal point, a leading plus or minus sign, and in the context of scientific notation, an additional plus or minus sign, an additional decimal point, and the letter E.

If *expr\$* is a string expression involving concatenation that contains more than 255 characters, the message ?STRING TOO LONG ERROR occurs. If the numeric value of *expr\$* exceeds the limits of real numbers, the message ?OVERFLOW ERROR occurs.

Not available in Integer BASIC.



# A

## Derived Numeric Functions

While the following list of derived functions is by no means complete, it does provide some of the most frequently needed formulas. Certain values of  $x$  will invalidate some functions (for example, if  $\cos(x)=0$  then  $\sec(x)$  is nonreal), so your program should check for them.

None of the derived functions will operate in Integer BASIC.

$$\text{ARCCOS}(x) = -\text{ATN}(x/\text{SQR}(-x*x+1))+1.5707633$$

Returns the inverse cosine of  $x$  ( $\text{ABS}(x) < 1$ ).

$$\text{ARCCOT}(x) = -\text{ATN}(x)+1.5707633$$

Returns the inverse cotangent of  $x$ .

$$\text{ARCCOSH}(x) = \text{LOG}(x+\text{SQR}(x*x-1))$$

Returns the inverse hyperbolic cosine of  $x$  ( $x \geq 1$ ).

$$\text{ARCCOTH}(x) = \text{LOG}((x+1)/(x-1))/2$$

Returns the inverse hyperbolic cotangent of  $x$  ( $\text{ABS}(x) > 1$ ).

$$\text{ARCCSC}(x) = \text{ATN}(1/\text{SQR}(x*x-1))+(\text{SGN}(x)-1)*1.5707633$$

Returns the inverse cosecant of  $x$  ( $\text{ABS}(x) > 1$ ).

$$\text{ARCCSCH}(x) = \text{LOG}((\text{SGN}(x)*\text{SQR}(x*x+1)+1)/x)$$

Returns the inverse hyperbolic cosecant of  $x$  ( $x > 0$ ).

$$\text{ARCSEC}(x) = \text{ATN}(\text{SQR}(x*x-1))+(\text{SGN}(x)-1)*1.5707633$$

Returns the inverse secant of  $x$  ( $\text{ABS}(x) \geq 1$ ).

$$\text{ARCSECH}(x) = \text{LOG}((\text{SQR}(-x*x+1)+1)/x)$$

Returns the inverse hyperbolic secant of  $x$  ( $0 < x \leq 1$ ).

$$\text{ARCSIN}(x) = \text{ATN}(x/\text{SQR}(-x*x+1))$$

Returns the inverse sine of  $x$  ( $\text{ABS}(x) < 1$ ).

$$\text{ARCSINH}(x) = \text{LOG}(x+\text{SQR}(x*x+1))$$

Returns the inverse hyperbolic sine of  $x$ .

$$\text{ARCTANH}(x) = \text{LOG}((1+x)/(1-x))/2$$

Returns the inverse hyperbolic tangent of  $x$  ( $\text{ABS}(x) < 1$ ).

$$\text{COSH}(x) = (\text{EXP}(x)+\text{EXP}(-x))/2$$

Returns the hyperbolic cosine of  $x$ .

$$\text{COT}(x) = 1/\text{TAN}(x)$$

Returns the cotangent of  $x$  ( $x < > 0$ ).

$$\text{COTH}(x) = \text{EXP}(-x)/(\text{EXP}(x)-\text{EXP}(-x))*2+1$$

Returns the hyperbolic cotangent of  $x$  ( $x < > 0$ ).

$$\text{CSC}(x) = 1/\text{SIN}(x)$$

Returns the cosecant of  $x$  ( $x < > 0$ ).

$$\text{CSCH}(x) = 2/(\text{EXP}(x)-\text{EXP}(-x))$$

Returns the hyperbolic cosecant of  $x$  ( $x < > 0$ ).

$$\text{LOG}_a(x) = \text{LOG}(x)/\text{LOG}(a)$$

Returns the base  $a$  logarithm of  $x$  ( $a > 0$ ,  $x > 0$ ).

$$\text{LOG}_{10}(x) = \text{LOG}(x)/2.30258509$$

Returns the common (base ten) logarithm of  $x$  ( $x > 0$ ).

$$\text{MOD } a(x) = \text{INT}((x/a - \text{INT}((x/a)) * a + .05) * \text{SGN}(x/a))$$

Returns  $x$  modulo  $a$ : the remainder after division of  $x$  by  $a$  ( $a < > 0$ ).

$$\text{SEC}(x) = 1/\text{COS}(x)$$

Returns the secant of  $x$  ( $x < > \pi/2$ ).

$$\text{SECH}(x) = 2/(\text{EXP}(x)+\text{EXP}(-x))$$

Returns the hyperbolic secant of  $x$ .

$$\text{SINH}(x) = (\text{EXP}(x)-\text{EXP}(-x))/2$$

Returns the hyperbolic sine of  $x$ .

$$\text{TANH}(x) = -\text{EXP}(-x)/(\text{EXP}(x)+\text{EXP}(-x))*2+1$$

Returns the hyperbolic tangent of  $x$ .

# B

## Editing Commands

This appendix summarizes the functions of the Apple II keystroke editing commands.

- Moves the cursor forward along the display line. Each character passed over is copied into memory as if it had been typed on the keyboard. Does not alter the screen display.
- ← Backspaces the cursor along the display line, erasing passed-over characters from memory but not from the display screen.
- REPT Causes a character printed by another key to be repeated as long as both keys are held down. The REPT key must be pressed *after* the other key.
- CTRL-X The Apple II disregards the current display line and moves the cursor to the left margin on the next line down.

### Esc Key Sequences

The following seven editing commands are two-key sequences. In each instance, press the Esc key, release it, and then press the second key of the sequence.

- Esc-A Moves the cursor one position to the right. Does not alter the screen display nor change memory.
- Esc-B Moves the cursor one position to the left. Does not alter the screen display nor change memory.
- Esc-C Moves the cursor down one line. Does not alter the screen display nor change memory.
- Esc-D Moves the cursor up one line. Does not alter the screen display nor change memory.
- Esc-E Deletes all characters from the cursor to the end of the display line.
- Esc-F Deletes all characters from the cursor to the end of the display screen.
- Esc-@ Clears the screen and moves the cursor to the upper left corner.

### Edit Mode Commands

The following four editing commands require the Autostart Monitor. They are effective only in *edit* mode. Enter edit mode by pressing the Esc key and leave edit mode by pressing any key other than the I, J, K, M, REPT, CTRL, or SHIFT keys.

- I Moves the cursor up one line without leaving edit mode.
- J Moves the cursor one position to the left without leaving edit mode.
- K Moves the cursor one position to the right without leaving edit mode.
- M Moves the cursor down one line without leaving edit mode.



# **C**

## **Error Messages**

Error messages are grouped into three categories: Integer BASIC, Applesoft, and DOS messages, and are listed alphabetically within each category.

DOS error messages and most Applesoft error messages have associated error codes. After an error has caused an ONERR GOTO branch to occur, the code for that error can be found in memory location 222. Table C-1, located at the end of this appendix, lists error messages by their code numbers.

### **INTEGER BASIC ERROR MESSAGES**

**\*\*\* > 255 ERR**

A value which should be between 0 and 255 is outside that range.

**\*\*\* > 32767 ERR**

A number greater than 32767 or less than -32767 has been entered or calculated.

**\*\*\* 16 FORS ERR**

More than 16 FOR loops are active.

**\*\*\* 16 GOSUBS ERR**

Seventeen more GOSUB statements than RETURN statements have been executed.

**\*\*\* BAD BRANCH ERR**

A branch to a nonexistent line number has been attempted.

**\*\*\* BAD NEXT ERR**

A NEXT with no matching FOR has been executed.

**\*\*\* BAD RETURN ERR**

More RETURN statements than GOSUB statements have been executed.

**\*\*\* DIM ERR**

The same array has been dimensioned more than once.

**\*\*\* MEM FULL ERR**

More memory is needed than is available.

**\*\*\* NO END ERR**

The very last instruction executed in a program was not END.

**\*\*\* RANGE ERR**

An array has been referenced with a subscript less than zero or greater than the array's size, or an argument in an HLIN, VLIN, PLOT, TAB, or VTAB instruction was outside the prescribed range.

**RETYPE LINE**

An error has been generated by an INPUT response. A diagnostic message is displayed first, and then this directive.

**\*\*\* STRING ERR**

An illegal string operation has been executed.

**\*\*\* STR OVFL ERR**

A string has been assigned more characters than it was dimensioned for.

**\*\*\* SYNTAX ERR**

An error in spelling, punctuation, or sequence, or any error not covered by another error message has occurred.

**\*\*\* TOO LONG ERR**

More than 12 parentheses have been nested or more than 128 characters have been entered in one line.

## **APPLESOFT ERROR MESSAGES**

**?BAD SUBSCRIPT ERROR**

An array has been referenced with the wrong number of subscripts or with one or more subscripts exceeding their dimensions. Error code 107.

**?CAN'T CONTINUE ERROR**

An attempt to continue (with the CONT command) was made when no program existed, a fatal error had occurred, or a change had been made to the program.

**?DIVISION BY ZERO ERROR**

An attempt has been made to divide by an expression that evaluates to zero. Error code 133.

**?FORMULA TOO COMPLEX ERROR**

More than two statements of the form IF *string* THEN have been executed. Error code 191.

**?ILLEGAL DIRECT ERROR**

An INPUT, DEF FN, or a GET command was entered in direct mode.

**?ILLEGAL QUANTITY ERROR**

A numeric value is outside the acceptable range for a string function, numeric function, graphics statement, and so forth. Error code 53.

**?NEXT WITHOUT FOR ERROR**

A NEXT with no matching FOR has been executed. A NEXT with no variable name generates this error only if there is no active FOR. Error code 0.

**?OUT OF DATA ERROR**

More DATA elements have been read than are available. Error code 42.

**?OUT OF MEMORY ERROR**

Can be caused by any of the following: program too large, too many variables, more than 10 levels of FOR loop nesting, more than 24 levels of subroutine nesting, more than 36 levels of parentheses nesting, LOMEM: set too high, or HIMEM: set too low. Error code 77.

**?OVERFLOW ERROR**

Too large or too small a number has been entered or calculated. The allowable range is approximately  $-1.7\text{E}+38$  to  $1.7\text{E}+38$ . Error code 69.

**?REDIM'D ARRAY ERROR**

A DIM statement for a previously dimensioned array has been executed. Most commonly occurs when an array was dimensioned by default. Error code 120.

**?RETURN WITHOUT GOSUB ERROR**

More RETURN statements than GOSUB statements have been executed. Error code 22.

**?STRING TOO LONG ERROR**

An attempt was made to concatenate strings totaling more than 255 characters. Error code 176.

**?SYNTAX ERROR**

An error in spelling, punctuation, or sequence, or any error not covered by another message has occurred. Error code 16.

**?TYPE MISMATCH ERROR**

A numeric expression or variable has been used where a string should be, or vice versa. Also occurs when the two sides of an assignment statement do not match in type. Error code 163.

**?UNDEF'D FUNCTION ERROR**

A user-defined function that has never been defined has been referenced. Error code 224.

**?UNDEF'D STATEMENT ERROR**

A branch to a nonexistent line number has been attempted. Error code 90.

## **DOS ERROR MESSAGES**

**DISK FULL**

An attempt has been made to store more information on a disk than it can hold. On a full disk this message may occur in place of a more appropriate message (e.g., FILE NOT FOUND.) Error code 9.

**END OF DATA**

An attempt has been made to read from a portion of a text file that has never been written to. Error code 5.

**FILE LOCKED**

An attempt has been made to use SAVE, BSAVE, WRITE, DELETE, or RENAME on a locked file. Error code 10.

**FILE NOT FOUND**

A file has been referenced that does not exist on the disk. This error only occurs if the DOS command that referenced the file does not create the file when it is not found. Error code 6.

**FILE TYPE MISMATCH**

A DOS command has referenced a file that is not of the required type. The LOAD, RUN, and SAVE commands may only be used with program files. The CHAIN command may only be used with an Integer BASIC program file. The OPEN, READ, WRITE, APPEND, POSITION, and EXEC commands may only be used with text files. The BLOAD, BSAVE, and BRUN commands may only be used with binary files. Error code 13.

**I/O ERROR**

An unsuccessful attempt to store to or retrieve from a disk has been made. Some common causes are: the disk drive door is open, the disk has not been initialized, no disk is in the drive, or the disk is defective. Error code 8.

**LANGUAGE NOT AVAILABLE**

An attempt to change languages with FP or INT has been made when the desired language was not in ROM or on the disk, or an attempt to load or RUN a program was made when the language of the program was similarly unavailable. Error code 1.

**NO BUFFERS AVAILABLE**

Another file buffer was required when all the available file buffers were already in use. Error code 12.

TABLE C-1. Error Codes

PEEK (222)	Error Description	Language
0	NEXT without FOR	Applesoft
1	Language not available	DOS
2 or 3	Range error	DOS
4	Write protected	DOS
5	End of data	DOS
6	File not found	DOS
7	Volume mismatch	DOS
8	I/O error	DOS
9	Disk full	DOS
10	File locked	DOS
11	Syntax error	DOS
12	No buffers available	DOS
13	File type mismatch	DOS
14	Program too large	DOS
15	Not direct command	DOS
16	Syntax error	Applesoft
22	RETURN without GOSUB	Applesoft
42	Out of DATA	Applesoft
53	Illegal quantity	Applesoft
69	Overflow	Applesoft
77	Out of memory	Applesoft
90	Undefined statement	Applesoft
107	Bad subscript	Applesoft
120	Redimensioned array	Applesoft
133	Division by zero	Applesoft
163	Type mismatch	Applesoft
176	String too long	Applesoft
191	Formula too complex	Applesoft
224	Undefined function	Applesoft
254	Bad response to an INPUT	Applesoft
255	CTRL-C has been struck	Applesoft

**NOT DIRECT COMMAND**

The following DOS commands may only be used from within PRINT statements in programmed mode: APPEND, OPEN, POSITION, READ, and WRITE. Error code 15.

**PROGRAM TOO LARGE**

A DOS command has attempted to put a file from the disk in the Apple II memory, and found insufficient memory to hold the file. Error code 14.

**RANGE ERROR**

A parameter used with a DOS command is outside of the range specified

for that parameter; for example, the D (drive) parameter must be either 1 or 2. Error code 2 or 3.

**SYNTAX ERROR**

A DOS command has an error in spelling, punctuation, or sequence. Error code 11.

**VOLUME MISMATCH**

The V (volume) parameter in a DOS command does not match the volume number of the disk accessed. Error code 7.

**WRITE PROTECTED**

An attempt has been made to use SAVE, BSAVE, or WRITE on a write-protected disk. Error code 4.

# D

## Intrinsic Subroutines

The following two tables list a number of useful machine language subroutines available on the Apple II. Table D-1 lists them by general function; it does not provide complete information about each subroutine. Find the entry point listed in Table D-1 in the first column of Table D-2 for details on registers affected, etc.

Table D-2 lists the subroutines in order by entry point. The third column shows which registers, if any, must contain specific values before the subroutine is executed. The fourth column shows which registers are affected by the execution of the subroutine.

Most of these subroutines have an equivalent in a BASIC command, or can be accessed from BASIC with a single CALL instruction. These equivalents appear in Table D-2. Some of the BASIC commands listed are only available in Applesoft; they are marked with an *A*.

Some subroutines, however, have no equivalent in either version of BASIC, and cannot be executed by a single CALL because one or more registers must be loaded with specific values prior to execution. Different techniques are required to handle this problem in the different versions of BASIC.

Integer BASIC provides a fairly simple solution. First execute a CALL -182 to place the current values of the registers in read/write memory. Then POKE the desired values in memory location 69 for the A register, 70 for the X register, and 71 for the Y register. Execute a CALL -193 to restore these values in the registers, and CALL the location of the subroutine you wanted to execute in the first place.

This technique does not work in Applesoft. You must instead write and execute a machine language subroutine that loads the registers with the desired values and then executes an assembly language JSR instruction to the entry point of the desired intrinsic subroutine.

TABLE D-1. Intrinsic Subroutines Cross-Referenced by Function

	Function	Entry Point
<b>Low-Resolution Graphics</b>	Plot a low-resolution graphics point.	\$F800
	Draw a low-resolution horizontal line.	\$F819
	Draw a low-resolution vertical line.	\$F828
	Clear all 48 low-resolution graphics rows to black (if in text mode, sets to inverse "@").	\$F832
	Clears the top 40 low-resolution rows to black (or inverse "@").	\$F836
	Increment the current low-resolution graphics color by three.	\$F85F
	Set low-resolution graphics color.	\$F864
	Read the color of a low-resolution graphics point.	\$F871
	Set low-resolution graphics mode, clear screen, and set four line text window.	\$FB40
<b>Input</b>	Wait for keystroke while flashing cursor, and seed random number generator at locations \$4E and \$4F.	\$FD1B
	Same as above except that escape codes are also allowed.	\$FD35
	Send carriage return to display screen, then allow input of an entire line of up to 256 characters.	\$FD67
<b>Output</b>	Send three blanks out to the currently selected output device.	\$F948
	Send from one to 256 blanks to the currently selected output device.	\$F94A
	Send a carriage return and line feed to the Apple II screen.	\$FC62
	Output a character to the currently selected output device.	\$FDED
	Output a character to the text window.	\$FDF0
<b>Bell Output</b>	Send BELL character (ASCII code 7) to the currently selected output device.	\$FBD9
	Beep the onboard speaker for 1/10 second.	\$FBE4
	Print the message ERR and beep the onboard speaker.	\$FF2D
	Beep the onboard speaker.	\$FF3A



TABLE D-1. Intrinsic Subroutines Cross-Referenced by Function (Continued)

	Function	Entry Point
<b>Text Window</b>	Set the Apple II screen to 24 rows by 40 columns.	\$FB2F
	Scroll the text window up one line.	\$FC70
<b>Cursor Control</b>	Send a backspace character to the screen, updating the cursor position.	\$FC10
	Move the cursor up one line. If the cursor is already at the top of the screen, it does not move.	\$FC1A
	Move the cursor down one line without changing its horizontal position.	\$FC66
	Scrolls the text window if the cursor is at the bottom of the screen.	
<b>Screen Clearing</b>	Clear the text window from the current cursor position to the lower righthand corner of the screen.	\$FC42
	Clear the text window from coordinates passed in registers to the lower righthand corner of the screen.	\$FC46
	Clear the entire text screen and move the cursor to the upper lefthand corner.	\$FC58
	Clear the text from the current cursor position to the end of the line.	\$FC9C
<b>Video Mode</b>	Set inverse video-mode (black on white).	\$FE80
	Set normal video mode (white on black).	\$FE84
<b>Print Register Contents</b>	Print Y and X register contents (in the format YYXX) on the currently selected output device.	\$F940
	Print A and X register contents (in the format AAXX) on the currently selected output device.	\$F941
	Print X register contents on the currently selected output device.	\$F944
	Print A register contents on the currently selected output device.	\$FDDA
<b>Move Register Contents</b>	Restore register contents (valid only if intrinsic routine at \$FF4A executed previously).	\$FF3F
	Save register contents in reserved Page Zero locations.	\$FF4A
<b>Misc.</b>	Read status of one paddle.	\$FB1E
	Execute a delay loop.	\$FCA8
	Return to BASIC, eliminating the program and variables in memory.	\$FEB0
	Entry point for the monitor.	\$FF69

TABLE D-2. Intrinsic Subroutines by Entry Point

Entry Point	Use	Registers to Load Before Calling	Registers Affected	BASIC Equivalent
\$F800	Plot a graphics point on low-resolution page 1.	Place row in A, column in Y.	None	PLOT
\$F819	Draw a low-resolution horizontal line.	Row in A, left column in Y, right column at memory location 44.	A,Y	HLIN
\$F828	Draw a low-resolution vertical line.	Column in Y, high row in A, low row at memory location 45.	None	VLIN
\$F832	Clear all 48 low-resolution graphics rows to black (if in text mode, sets to all "@").	None	A,Y	CALL - 1998
\$F836	Clear the low-resolution graphics rows, leaving the text window intact.	None	A,Y	GR (see \$FB40)
\$F85F	Increment the current low-resolution graphics color by three.	None	A	CALL - 1985
\$F864	Set low-resolution graphics color.	Color number in A.	A	COLOR
\$F871	Read the color of a low-resolution graphics point.	Row in A, column in Y.	A (contains color number)	SCRN
\$F940	Print Y and X register contents (in the format YYXX) on the screen or other output device selected.	None	None	CALL - 1728
\$F941	Print A and X registers (AAXX) as above.	None	None	CALL - 1727
\$F944	Print X register contents.	None	None	CALL - 1724
\$F948	Send three blanks out to the currently selected output device (determined by CSW contents).	None	X,A	CALL - 1720
\$F94A	Send 1 to 256 blanks to the currently selected output device.	Number of blank spaces in A; (loading 0 prints 256 blanks).	None	SPC ( ) <sup>A</sup> CALL - 1718
\$FB1E	Read status of paddle 0, 1, 2, or 3.	Paddle number in X.	0-FF in Y register. A contents destroyed.	PDL ( )
\$FB2F	Set the Apple II text screen to 40 rows by 24 columns.	None	A	TEXT
\$FB40	Set low-resolution graphics mode, clear screen and set 4-line text window.	None	A,Y	GR
\$FBD9	Send BELL character (ASCII code 7) to the current output device.	None	A,Y	CALL - 1063
\$FBE4	Beep Apple II speaker for 1/10 second.	None	A,Y	CALL - 1052
\$FC10	Send a backspace character to the screen, updating cursor position.	None	A	CALL - 1008
\$FC1A	Move the cursor up one line. If already at the top of the screen, cursor does not move.	None	A	CALL - 998
\$FC42	Clear the text window from the present cursor position to the lower righthand corner of the screen.	None	A,Y	CALL - 958
\$FC46	Clear the text window from coordinates passed in registers to the lower righthand corner of the screen.	Column in Y, row in A.	A,Y	CALL - 954
\$FC58	Clear the entire text screen and move the cursor to the upper lefthand corner.	None	A,Y	HOME <sup>A</sup> CALL - 936

<sup>A</sup> Denotes BASIC commands available in Applesoft only.

TABLE D-2. Intrinsic Subroutines by Entry Point (Continued)

Entry Point	Use	Registers to Load Before Calling	Registers Affected	BASIC Equivalent
\$FC62	Send a carriage return and line feed to the Apple II screen.	None		CALL -926
\$FC66	Move the cursor down one line without changing its horizontal position. Scrolls text up one line if cursor is at the bottom of the screen.	None	A, Y	CALL -922
\$FC70	Scroll the text window up one line.	None	A, Y	CALL -912
\$FC9C	Clear text from the current cursor position to the end of the line. Cursor position remains unchanged.	None	A, Y	CALL -868
\$FCA8	Execute a delay loop which is $0.5(5x^2 + 27x + 26)$ microseconds long.	Delay value (x) in A.	A	CALL -856
\$FD1B	Wait for keystroke; flash cursor while waiting. Seed random number generator at memory locations 78 and 79.	None	Character returned in A. X, Y	CALL -756
\$FD35	Same as \$FD1B, except that escape codes are also allowed.	None	Character returned in A. X, Y	CALL -715
\$FD67	Send carriage return to screen; allow input of an entire line of data, up to 256 characters.	Prompt character at memory location 51.	Y, A. X contains length of entry. Data entered starts at memory location \$200	INPUT
\$FDDA	Print the value in the accumulator as two hexadecimal digits.	Data in A.	A	CALL -550
\$FDED	Output a character to the currently selected output device.	Character in A.	None	PRINT
\$FDFO	Output a character to the Apple text window.	Character in A.	None	PRINT
\$FE80	Set inverse video (black-on-white text).	None	Y	INVERSE <sup>A</sup> CALL -384
\$FE84	Set normal video mode (white-on-black text).	None	Y	NORMAL <sup>A</sup> CALL -380
\$FE80	Return to BASIC, eliminating the program and variables in memory.	None	A, X, Y	CALL -336
\$FF20	Print the message ERR and beep onboard speaker.	None	A	CALL -211
\$FF3A	Beep the onboard speaker.	None	A	CALL -198
\$FF3F	Restore register contents (valid only if intrinsic routine at \$FF4A executed previously).	None	Register contents restored from these locations: A register: 69 (\$45) S register: 72 (\$48) X register: 70 (\$46) Stack Pointer: 73 (\$49) Y register: 71 (\$47)	CALL -193
\$FF4A	Save register contents in reserved Page 0 locations: A register: 69 (\$45) S register: 72 (\$48) X register: 70 (\$46) Stack Pointer: 73 (\$49) Y register: 71 (\$47)	None	None	CALL -182
\$FF69	Entry point for the monitor.	None	None	CALL -151

<sup>A</sup> Denotes BASIC commands available in Applesoft only.



# E

## Useful PEEK and POKE Locations

Each of the memory locations listed below is expressed in terms of a decimal number less than 32767 in magnitude. Memory locations above 32767 are expressed in terms of a negative number. There is a positive number which refers to the same location. Add 65536 to the listed negative location to get the positive equivalent (e.g.,  $65536 - 16384 = 49152$ ).

Some of the functions described below are actuated by just accessing them. This means that any time a PEEK statement accesses the specified memory location, the indicated action takes place. A POKE statement to the specified memory location also triggers the action, but because of the operating characteristics of the microprocessor in the Apple II, a POKE statement actually triggers the action *twice*. In this case, POKE is the same as two PEEK statements. Usually this makes no difference, but in cases like -16336 (Speaker Click) it does. The value placed in memory by the POKE statement is irrelevant in such address-actuated actions.

### TEXT WINDOW AND CURSOR CONTROL LOCATIONS

#### 32 Left Margin of the Text Window

Specifies the column of the left text window margin. PEEK returns a value in the range 0 through 39, 0 being the left edge of the screen. Changing this location does not affect the width of the text window; the left and right margins both move.

If you POKE a value greater than 39 in this location, or if the value of this location plus the width of the text window exceeds 40, some or all of the output meant for the screen will be put in memory outside the screen area. This could destroy part of your program or other essential data.

### 33 Text Window Width

Specifies the width of the text window. The value in this location must be in the range 1 through 40. Changing this location sets the right margin at the column that is the specified number of characters away from the left margin (memory location 32).

A value of zero in this location (i.e., a width of zero) can destroy the BASIC interpreter. If you POKE a value greater than 40 into this location, or if the value in this location plus the value in location 32 (left margin) exceeds 40, some or all of the output meant for the screen will be put in memory outside the screen area. This could destroy part of your program or other essential data.

### 34 Top Margin of the Text Window

Specifies the top margin of the text window. The value in this location must be in the range 0 through 23; 0 specifies the top row on the screen, 23 the bottom. If you POKE a value greater than 23 into this location, some or all of the output meant for the screen will go into memory outside the screen area, wiping out data that could be important. Do not set the top margin of the text window below the bottom margin.

### 35 Bottom Margin of the Text Window

Specifies the bottom margin of the text window. The value in this location must be in the range 0 through 23; 0 specifies the top row on the screen, 23 the bottom. If you POKE a value greater than 23 into this location, some or all of the output meant for the screen will go into memory outside the screen area, wiping out data that could be important. Do not set the bottom margin of the text window above the top margin.

### 36 Horizontal Position of the Cursor

Specifies the current horizontal position of the cursor. PEEK returns a value in the range 0 through 39; it specifies the cursor's position relative to the left margin of the text window (not necessarily the left edge of the screen). This location can be used to position beyond the right edge of the text window (and subsequently print there with PRINT), but the cursor only stays there long enough to print one character. Do not put a value in this location that, when added to the left screen margin (location 32), exceeds 39.

This PEEK is equivalent to the Applesoft function POS.

### 37 Vertical Position of the Cursor

Specifies the current vertical position of the cursor. PEEK returns a value in the range 0 through 23, relative to the top of the screen (not the top of the text window). Do not put a value over 23 in this location.

## ERROR HANDLING LOCATIONS

### 216 Error Flag

Indicates whether an ONERR GOTO is in effect. If bit 7 of this memory location is 1 (i.e., if this location has a value of 128 or more), an ONERR GOTO statement has been encountered, and control will branch to the line number specified when an error occurs. POKE a value less than 128 to disable a previously executed ONERR GOTO statement.

### 218 and 219 Error-Causing Line Number

When an error triggers a branch according to an ONERR GOTO statement, these locations specify the line number in which the error occurred. This line number is  $\text{PEEK}(219) * 256 + \text{PEEK}(218)$ .

### 222 Error Type Code

Specifies which type of error has occurred. The error codes and their descriptions are given in Appendix C.

## KEYBOARD LOCATIONS

### –16384 Character from Keyboard

Reads the keyboard. If the value in this location is greater than 127 (i.e., if bit 7 is 1), a key has been pressed. Determine the ASCII code of the key last pressed by subtracting 128 from this value.

### –16368 Keyboard Flag

Resets keyboard strobe (bit 7 of location –16384) to zero so that the next character may be read in.

## “CLICK” OUTPUT LOCATIONS

### –16352 Cassette Click

Generates an audible click on the cassette output jack.

### –16336 Speaker Click

Generates a click on the internal speaker.

## DISPLAY SWITCHES

The memory locations listed in this section set certain switches which determine display characteristics. There are no real physical switches; only PEEK and POKE commands affect the settings. There are four switches which can each be set in two different positions, as shown in Figure E-1. With text mode selected, the only other switch that has any effect is the Page 1/Page 2 switch.

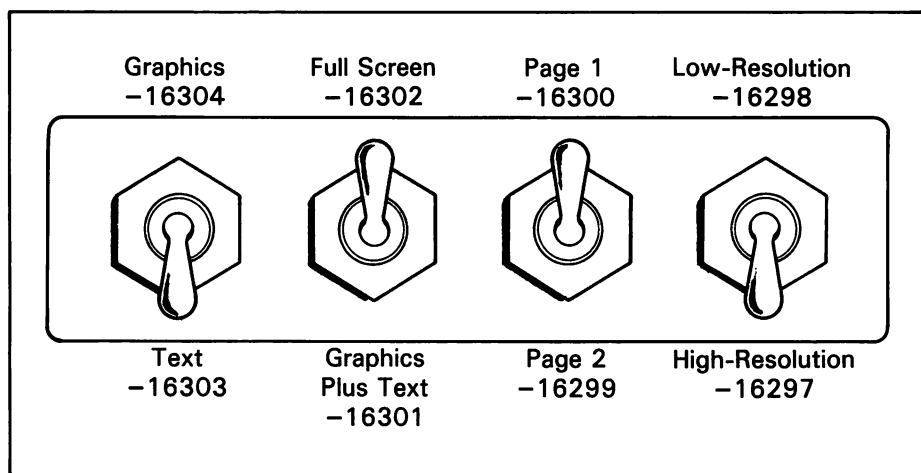


FIGURE E-1. PEEK/POKE Graphics and Text Locations

**-16304 Select Graphics Mode**

Selects graphics mode. The graphics screen is not cleared to black. The graphics mode may be low or high resolution, page 1 or page 2, full-screen graphics or mixed graphics and text. These characteristics are determined by other memory locations.

**-16303 Select Text Mode**

Selects text mode. The text may be from either page 1 or page 2; this is determined by other memory locations.

**-16302 Select Full-Screen Graphics**

Selects full-screen graphics. If the screen is in text mode, this will not be visible until location -16304 is accessed.

**-16301 Select Graphics Plus Text**

Establishes a four-line text window at the bottom of the screen. If the screen is in text mode, this will not be visible until location -16304 is accessed.

**-16300 Select Screen Page 1**

Selects graphics or text page 1.

**-16299 Select Screen Page 2**

Selects graphics or text page 2.

**-16298 Select Low-Resolution Graphics**

Selects low-resolution graphics. If the screen is in text mode, this will not be visible until location -16304 is accessed.

**-16297 Select High-Resolution Graphics**

Selects high-resolution graphics. If the screen is in text mode, this will not be visible until location -16304 is accessed.



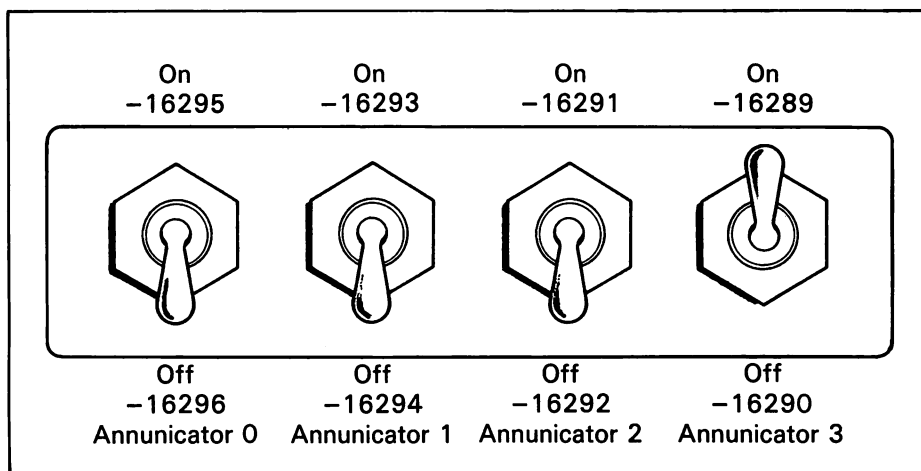


FIGURE E-2. Game Control Outputs (Annunciators) Manipulation

## GAME CONTROL LOCATIONS

The memory locations in this section turn game control outputs on or off, sense whether pushbuttons are being pressed or not, and actuate a strobe output. Figure E-2 shows how the game control outputs are manipulated.

All inputs and outputs for these PEEK and POKE statements connect to the game control connector, pictured in Figure E-3.

### -16296 Annunciator 0 Off

Turns off game control output (annunciator) number 0. The voltage on pin 15 of the game control connector is set to approximately 0 volts (TTL high).

### -16295 Annunciator 0 On

Turns on game control output (annunciator) number 0. The voltage on pin 15 of the game control connector is set to approximately +5 volts (TTL low).

### -16294 Annunciator 1 Off

Turns off game control output (annunciator) number 1. The voltage on pin 14 of the game control connector is set to approximately 0 volts (TTL high).

### -16293 Annunciator 1 On

Turns on game control output (annunciator) number 1. The voltage on pin 14 of the game control connector is set to approximately +5 volts (TTL low).

### -16292 Annunciator 2 Off

Turns off game control output (annunciator) number 2. The voltage on pin 13 of the game control connector is set to approximately 0 volts (TTL high).

### -16291 Annunciator 2 On

Turns on game control output (annunciator) number 2. The voltage on pin

- 13 of the game control connector is set to approximately +5 volts (TTL low).
- 16290 Annunciator 3 Off  
Turns off game control output (annunciator) number 3. The voltage on pin 12 of the game control connector is set to approximately 0 volts (TTL high).
  - 16289 Annunciator 3 On  
Turns on game control output (annunciator) number 3. The voltage on pin 12 of the game control connector is set to approximately +5 volts (TTL low).
  - 16287 Read Pushbutton 0  
When the pushbutton on game control number 0 is being pressed, the value in this location exceeds 127; when it is not being pressed, the value is 127 or less. Pushbutton 0 connects to pin 2 of the game control connector.
  - 16286 Read Pushbutton 1  
When the pushbutton on game control number 1 is being pressed, the value in this location exceeds 127; when it is not being pressed, the value is 127 or less. Pushbutton 1 connects to pin 3 of the game control connector.
  - 16285 Read Pushbutton 2  
When the pushbutton on game control number 2 is being pressed, the value in this location exceeds 127; when it is not being pressed, the value is 127 or less. Pushbutton 2 connects to pin 4 of the game control connector.
  - 16272 Strobe Output  
Normally pin 5 of the game control connector is +5 volts. If you PEEK memory location -16285, it drops to 0 volts for one-half microsecond. POKE will trigger the strobe twice.

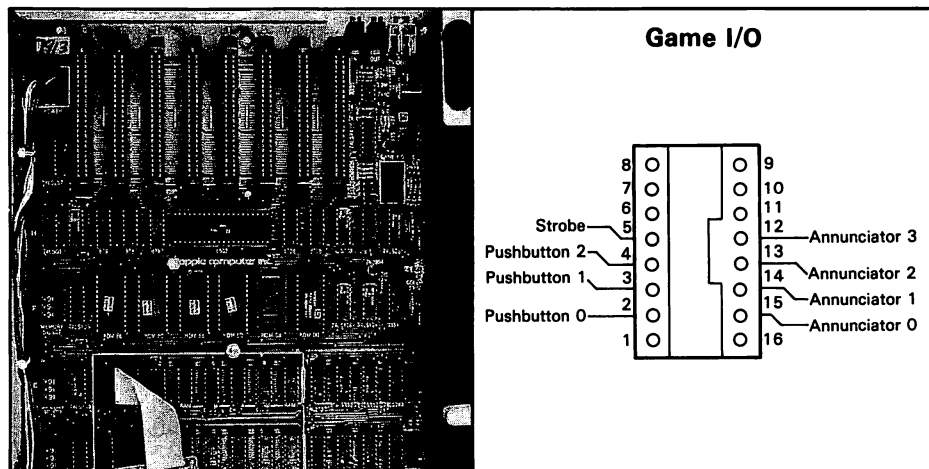


FIGURE E-3. Game Control Inputs and Outputs

# F

## BASIC Reserved Words

The Apple II interprets every occurrence of the following reserved words as a BASIC command, statement, or function. The only exception is when they are part of text strings enclosed in quotation marks. So keep reserved words out of your variable names. Watch especially for the short reserved words.

You may enter reserved words with embedded blank spaces; the Apple II will compress the blanks out.

### INTEGER BASIC

ABS	END	LET	PDL	SAVE
AND	FOR	LIST	PEEK	SCRN
ASC	GOSUB	LOAD	PLOT	SGN
AT	GOTO	LOMEM:	POKE	STEP
AUTO	GR	MAN	POP	TAB
CALL	HIMEM:	MOD	PRINT	TEXT
COLOR=	HLIN	NEW	PR#	THEN
CON	IF	NEXT	REM	TO
DEL	IN#	NOT	RETURN	TRACE
DIM	INPUT	NOTRACE	RND	VLIN
DSP	LEN	OR	RUN	VTAB

**APPLESOFT**

Reserved words in Applesoft are tokenized: each word takes up only one byte of program storage. The tokens are listed with each reserved word below. They are also listed in numerical order in Appendix I.

Applesoft will not recognize the reserved word TO properly if:

1. The first nonblank character before TO is the letter A, and
2. One or more blanks separate the T and O.

ABS	(212)	HTAB	(150)	REM	(178)
AND	(205)	IF	(173)	RESTORE	(174)
ASC	(230)	IN#	(139)	RESUME	(166)
AT	(197)	INPUT	(132)	RETURN	(177)
ATN	(225)	INT	(211)	RIGHT\$	(233)
CALL	(140)	INVERSE	(158)	RND	(219)
CHR\$	(231)	LEFT\$	(232)	ROT=	(152)
CLEAR	(189)	LEN	(227)	RUN	(172)
COLOR=	(160)	LET	(170)	SAVE	(183)
CONT	(187)	LIST	(188)	SCALE=	(153)
COS	(222)	LOAD	(182)	SCRN(	(215)
DATA	(131)	LOG	(220)	SGN	(210)
DEF	(184)	LOMEM:	(164)	SHLOAD	(154)
DEL	(133)	MID\$	(234)	SIN	(223)
DIM	(134)	NEW	(191)	SPC(	(195)
END	(128)	NEXT	(130)	SPEED=	(169)
EXP	(221)	NORMAL	(157)	SQR	(218)
FLASH	(159)	NOT	(198)	STEP	(199)
FN	(194)	NOTRACE	(156)	STOP	(179)
FOR	(129)	ON	(180)	STORE	(168)
FRE	(214)	ONERR	(165)	STR\$	(228)
GET	(190)	OR	(206)	TAB(	(192)
GOSUB	(176)	PDL	(216)	TAN	(224)
GOTO	(171)	PEEK	(226)	TEXT	(137)
GR	(136)	PLOT	(141)	THEN	(196)
HCOLOR=	(146)	POKE	(185)	TO	(193)
HGR	(145)	POP	(161)	TRACE	(155)
HGR2	(144)	POS	(217)	USR	(213)
HIMEM:	(163)	PRINT	(186)	VAL	(229)
HLIN	(142)	PR#	(138)	VLIN	(143)
HOME	(151)	READ	(135)	VTAB	(162)
HPlot	(147)	RECALL	(167)	WAIT	(181)
				XDRAW	(149)

**DOS**

DOS commands are only considered reserved words if they are used in immediate mode or in a PRINT statement that begins with a CTRL-D character (ASCII code 4).

APPEND	CHAIN	INIT	POSITION	SAVE
BLOAD	CLOSE	LOAD	READ	UNLOCK
BRUN	DELETE	LOCK	RENAME	VERIFY
BSAVE	EXEC	OPEN	RUN	WRITE



# G

## Memory Usage

### GENERAL MEMORY ORGANIZATION

The Apple II memory is divided into three general categories: read/write memory (also called random access memory or RAM), read-only memory (ROM), and input/output locations (I/O). Memory locations 0 through 49151 (\$BFFF hexadecimal) are in RAM, locations 49152 (\$C000) through 53247 (\$CFFF) are in ROM. Your system does not necessarily have actual memory for all of these locations. For instance, if you have only 16K of RAM, memory locations 16384 (\$4000) through 49151 (\$BFFF) are not usable.

Table G-1 shows how memory is allocated on an Apple II system. Notice there are two blocks of free memory locations, surrounding the two high-resolution graphics pages. The system pointer LOMEM: keeps track of the lower bound on this free area, and the system pointer HIMEM: marks the upper end. This read/write memory can be used for a number of things. Among them are the nonfirmware Applesoft interpreter (from cassette or disk), the Disk Operating System (DOS), high-resolution graphics, and your BASIC program and its variables.

TABLE G-1. BASIC Memory Organization

Location		Type of Memory	Usage
Decimal	Hex		
0-255	\$0-\$OFF	RAM	System programs
256-511	\$100-\$1FF	RAM	System stack
512-767	\$200-\$2FF	RAM	Keyboard input buffer
768-1023	\$300-\$3FF	RAM	Monitor vector locations
1024-2047	\$400-\$7FF	RAM	Text and low-resolution graphics page 1
2048-3071	\$800-\$BFF	RAM	Text and low-resolution graphics page 2
3072-8191	\$C00-\$1FFF	RAM	Free
8192-16383	\$2000-\$3FFF	RAM	High-resolution graphics page 1
16384-24575	\$4000-\$5FFF	RAM	High-resolution graphics page 2
24576-49151	\$6000-\$BFFF	RAM	Free
49152-49279	\$C000-\$C07F	I/O	Special built-in locations
49280-49407	\$C080-\$C0FF	I/O	Peripheral card I/O space
49408-51199	\$C100-\$C7FF	I/O	Peripheral card memory
51200-53247	\$C800-\$CFFF	I/O	Peripheral card expansion memory
53248-65535	\$D000-\$FFFF	ROM	Integer BASIC, Applesoft, the Monitor or the Autostart Monitor, etc.

## THE BASIC LANGUAGE INTERPRETERS

As you can see in Table G-1, the Integer BASIC interpreter always resides in ROM. The Applesoft interpreter also resides in ROM if your system has the Applesoft Firmware card or the Language System installed. Otherwise, the Applesoft interpreter occupies approximately 10K bytes of memory starting at 2048 (\$800).

## DOS MEMORY REQUIREMENTS

You need at least 16K of memory to use DOS. When booted, DOS takes up approximately 10K at the top of memory. HIMEM: is set just below the memory used by DOS. Figure G-1 shows which sections of memory are used by DOS on various sizes of systems. Note that you must have at least 24K of RAM to support both DOS and disk (or cassette) Applesoft, and at least 32K to use page 1 of high-resolution graphics with DOS. Figure G-1 also clearly shows the conflict between disk (or cassette) Applesoft and high-resolution graphics page 1.

DOS uses several additional sections of memory while it is booting (see Figure G-2). Anything in those areas before booting will be gone after booting.

## INTEGER BASIC MEMORY USAGE

Integer BASIC program lines reside in the high end of free read/write memory, starting at HIMEM:. As shown in Figure G-3, HIMEM: is automatically adjusted as you add, delete, and change program lines.



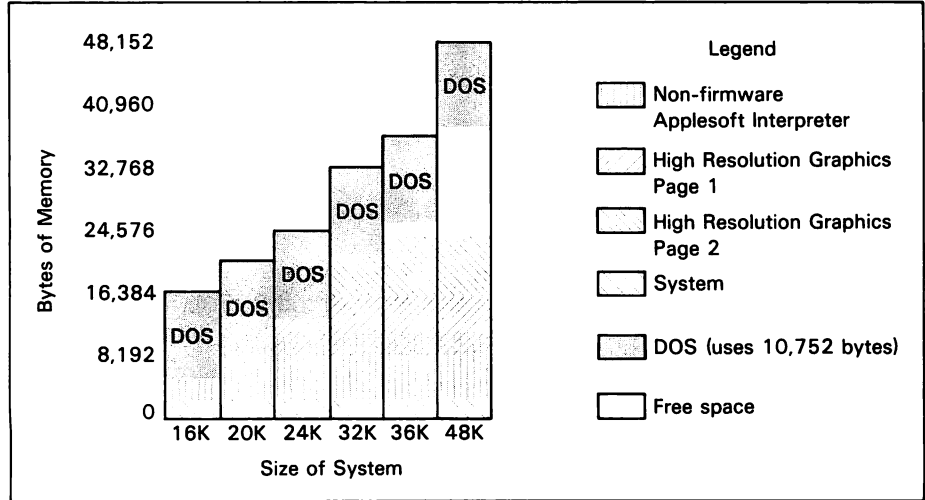


FIGURE G-1. Read/Write Memory (RAM) Usage

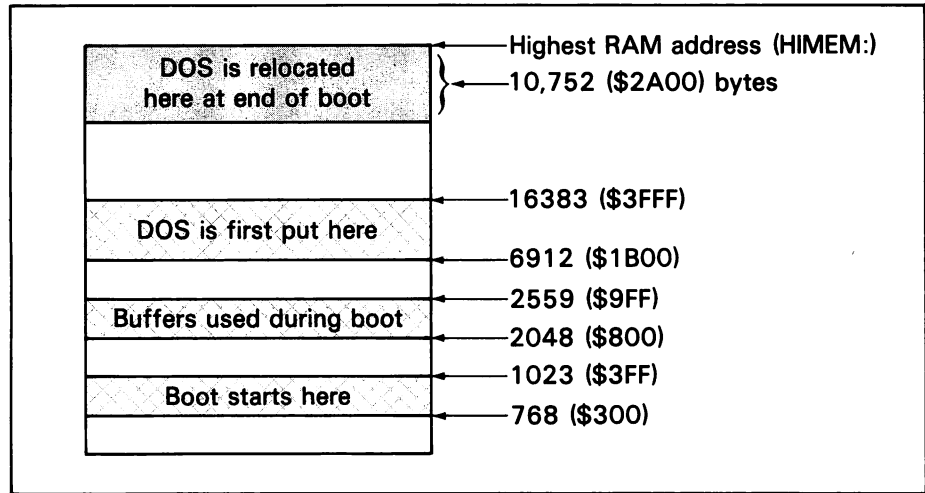


FIGURE G-2. Memory Used During DOS Boot

Variables are stored starting at LOMEM: and working up. As variable storage requirements change, LOMEM: adjusts automatically. Each numeric variable is mapped into memory with four attributes: the variable name, the DSP on/off byte, the memory location of the next variable, and the actual value or values of the variable.

The variable name may be up to 100 characters long. Each character is represented in memory by its ASCII code, with the high-order bit set to 1.

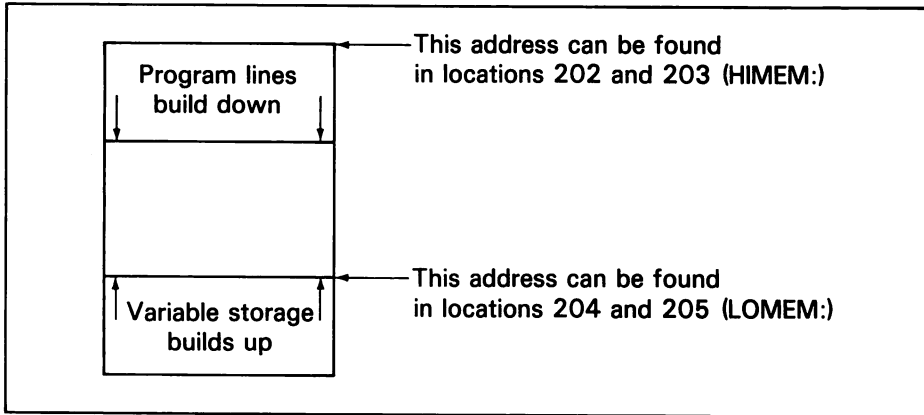


FIGURE G-3. Integer BASIC Program Memory Map

The DSP byte indicates whether the BASIC command DSP is in force for this variable. This byte, normally 0, is set to 1 when DSP is executed for this variable, and reset to 0 when NO DSP is executed.

The address of the next variable is stored in two bytes, low-order first.

The data is stored in pairs of bytes, low-order byte first. If the variable is not an array, there is only one such pair. If the variable is an array, there is one pair per element, listed in order starting with element zero. There is no distinction between a simple variable and an array with the same name; the simple variable is the zero element of the array.

String variables are stored similarly. The variable name, DSP byte, and next variable address are stored in the same fashion as numeric variables. The ASCII code for each character of the string takes one byte, with the high-order bit set to 1. The last character of the string is followed by a string terminator byte, in which the high-order bit is 0.

## APPLESOFT MEMORY USAGE

Applesoft program lines occupy the low end of free read/write memory starting at LOMEM:, as shown in Figure G-4. As you add, delete, and change program lines, LOMEM: adjusts automatically. Simple numeric variables and string pointers are stored directly above the program lines. Arrays and string array pointers are stored above the simple variables. String *values* are stored at the top of memory, starting at HIMEM:. As you use more string values, HIMEM: automatically adjusts downward.

Each numeric variable and string pointer uses seven bytes of memory. Each real variable uses two ASCII codes (two bytes) for the variable name (both with the high-order bit set to 0). The value is stored in scientific notation with one byte for the exponent and four bytes for the mantissa. The bytes of the mantissa are in order from the most significant byte to the least significant byte.

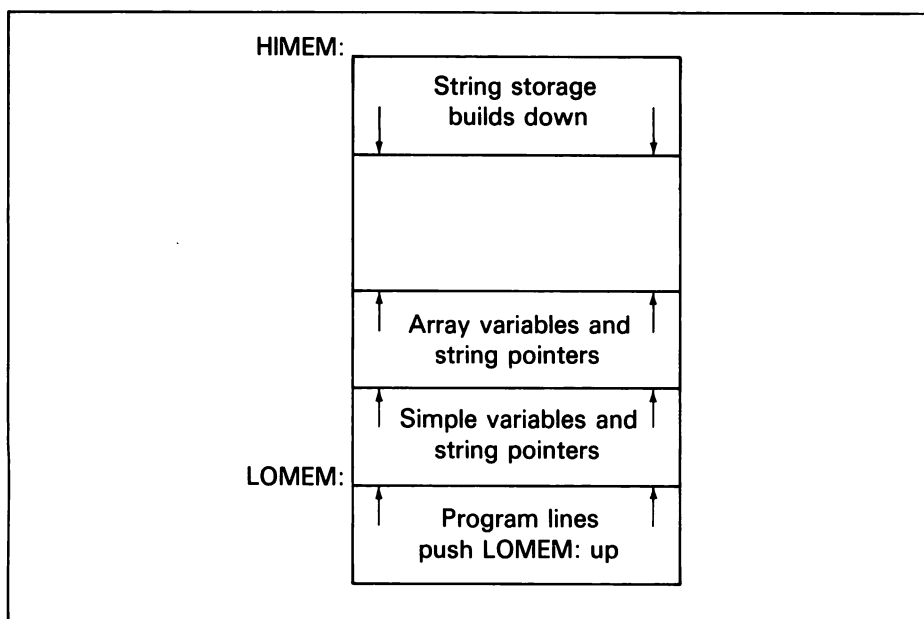


FIGURE G-4. Applesoft Program Memory Map

Each integer variable also uses two ASCII codes (two bytes) for the variable name (both with the high-order bit set to 1) and two bytes for the value of the variable, high-order byte first.

Each string pointer uses two ASCII characters (two bytes) for the variable name (the high-order bit of the first is 1, that of the second is 0), one byte for the length of the string, and two bytes for the address of the string value, low-order byte first. The last three bytes of an integer variable and the last two of a string pointer are unused.

Numeric arrays and string pointer arrays are stored immediately above the variables. The name of the variable is stored as an ASCII code in the first two bytes; both high-order bits are 0 for real variables, both are 1 for integer variables, and the first is 1 and the second 0 for string pointers.

The variable name is followed by two bytes indicating the location of the next variable. This is given relative to the first byte of this variable name, low-order byte first. Next is one byte for the number of dimensions, then two bytes per dimension (high-order first) indicating the size of each dimension. The sizes are listed in reverse order, i.e., the size of the first dimension is shown last.

Each element of the array is then listed, from element (0,0,...,0) to element (N,N,...,N). The elements are stored in order, with the leftmost index incremented first. Each real array element uses five bytes, one for the exponent, and four (most significant first) for the mantissa. Each integer element uses two bytes, high-order first. Each string pointer element uses three bytes, one for the length of the string and two (low-order first) for the address of the string.

String values are stored at the high end of free read/write memory. They require one byte of memory per character. Duplicate strings are only stored once; two or more string pointers can point to one location. As new string values are created, they are placed in the next available free space (HIMEM: adjusts downward). Strings that are no longer being used stay in memory. The FRE function forces a *garbage collection*, eliminating all abandoned strings and resetting HIMEM:.

# H

## Disk II Format

Information is stored on the diskette in 35 concentric bands, called *tracks*. These tracks are numbered 0 through 34 (\$0 through \$22). Each track is divided into 16 segments, called *sectors*, numbered 0 through 15 (\$0 through \$F). Each sector can hold up to 256 bytes of data. There are a total of 455 sectors on the disk, holding up to 116,480 bytes of data.

DOS transfers data to and from the disk one sector at a time. It uses two 256-byte file buffers in memory, one for reading and one for writing, for each active file.

Each type of file (text, program, and binary) has its own format on the disk. Text files are stored in ASCII code, one byte per character. A zero byte marks the end of the file. All bytes of a text file are interpreted as text.

The first two bytes of the first sector of a BASIC program file indicate program length, low-order byte first. The remainder of the file contains the program, in ASCII code. In an Applesoft file, reserved words are tokenized rather than spelled out (a token is a single ASCII code byte). See Appendix F for a list of tokens in alphabetical order and Appendix I for a list in numerical order.

The first two bytes of the first sector of a binary file show the starting address for the binary data in read/write memory, low-order byte first. The next two bytes show the length of the file, low-order byte first. The remainder of the file contains binary data.

## THE TRACK/SECTOR LIST

DOS normally writes to the disk wherever it can find a free sector. This means that one multisector file may be scattered over several tracks. DOS makes a list of the track and sector numbers used by each file and stores it in one or more additional sectors on the diskette. This is called the *track/sector list*.

Each sector of the track/sector list contains a pointer to the next sector of the track/sector list (if any) and points to as many as 122 file-contents sectors.

If a sector of a file is unused, the pointer to it in the track/sector list is 0. If an entire sector at the beginning of the track/sector list has zero pointers, that sector is not stored on the disk. Thus, if record number 5000 is the first and only record in a random-access file that has an L parameter of 256 (one record per sector), only two diskette sectors will be used: one for the data in record number 5000, and one for the 41st sector of the track/sector list.

Byte 0 and bytes 3 through 12 of the track/sector list sector are not used. Bytes 1 and 2 contain the track and sector numbers, respectively, of the next sector of the list. If these bytes are both 0, this is the last sector of the list.

## THE DIRECTORY

DOS uses track 17 (\$11) for the diskette directory. For each file, the directory contains the name of the file, the file type, the number of sectors occupied by the file (modulo 256), and the location of the file's track/sector list. Most of this information is displayed on the screen by the CATALOG command.

Each sector of the directory contains information for up to seven files. The directory begins in track 17, sector 15. When this sector is filled up, the directory continues in sector 14, and so on, through sector 1. The directory can contain entries for up to 84 files.

Byte 0 and bytes 3 through 10 of each directory sector are not used. Bytes 1 and 2 contain, respectively, the track and sector numbers of the next sector of the directory. If both are 0, this sector is the last in the directory. Bytes 11 through 255 contain the directory entries. Each entry takes 35 bytes; the first entry is in bytes 11 through 45, the second is in bytes 46 through 80, etc.

Directory entries are all written in the same format. Table H-1 itemizes the contents of each entry. Table H-2 explains how the file type is encoded in each directory entry.

Sector 0 of track 17 does not have directory entries. Instead, it holds identification status, physical description, and space availability information for the disk. Table H-3 outlines the contents of this important sector, called the *Volume Table of Contents*.

Each four-byte group from byte 56 through 195 of the Volume Table of Contents contains an availability map for one of the disk's tracks. Each map identifies which sectors of the associated track are in use and which are available. A bit has a value of 0 when the corresponding sector is in use. It has a value of 1 when the corresponding sector is free. Table H-4 shows which bytes identify which sectors.

TABLE H-1. Directory Entry Format

Relative Byte Number	Contents of Byte
0	Track number of the file's track/sector list. Changed to 255 when the file is deleted (former contents retained in relative byte 34).
1	Sector number of the file's track/sector list.
2	File type. See Table H-2.
3-32	File name, in ASCII.
33	Number of sectors used by the file, modulo 256.
34	End mark. Normally 0, but changed to the former contents of relative byte 0 when the file is deleted.

TABLE H-2. Disk Directory File Type Encoding

Bit	Feature
0	File is an Integer BASIC program file if this bit is 1.
1	File is an Applesoft program file if this bit is 1.
2	File is a binary file if this bit is 1.
3-6	Reserved for future expansion.
7	File is locked if this bit is 1.
If bits 0 through 6 are all 0, file is a text file.	

TABLE H-3. Volume Table of Contents (Sector 0, Track 17)

Byte	Description
0	Not used
1	Track number of first directory sector
2	Sector number of first directory sector
3	DOS release number
4-5	Not used
6	Diskette volume number
7-38	Not used
39	Maximum number of track/sector pairs possible in each sector of a track/sector list
40-47	Not used
48-51	Mask for the sector availability maps
52	Number of tracks per diskette
53	Number of sectors per diskette
54-55	Number of bytes per sector: low-order byte in 54, high-order byte in 55
56-59	Sector availability map, track 0
60-63	Sector availability map, track 1
64-195	Sector availability maps, tracks 2 through 195
196-255	Not used

TABLE H-4. Sector Availability Map

Byte	Bit	Sector
First	7	12
	6	11
	5	10
	4	9
	3	8
	2	7
	1	6
	0	5
Second	7	4
	6	3
	5	2
	4	1
	3	0
	2-0	Unused
Third	All	Unused
Fourth	All	Unused



# I

## **ASCII Character Codes and Applesoft Reserved Word Tokens**

The first table in this appendix shows ASCII codes 1 through 96 and the characters they represent. ASCII codes in the range 96 through 127 produce the same characters on the Apple II display screen as codes 64 through 95, although on some other output devices codes 96 through 127 produce lower-case letters. No keystrokes generate codes 96 through 127.

ASCII codes 128 through 255 repeat codes 0 through 127. No keystroke will generate them.

The second table in this appendix lists the Applesoft reserved words. Each reserved word takes up only one byte in program memory. Each reserved word is represented by a code, called a token, in the range 128 through 255. The token replaces the spelled-out reserved word in the Apple II memory and on the disk. The list is in numerical order by token. Appendix F contains a list of reserved words in alphabetical order.

## ASCII Character Codes

ASCII Code	Display Screen Character	Keystroke	ASCII Code	Display Screen Character	Keystroke
0		CTRL-@	48	0	0
1		CTRL-A	49	1	1
2		CTRL-B	50	2	2
3		CTRL-C	51	3	3
4		CTRL-D	52	4	4
5		CTRL-E	53	5	5
6		CTRL-F	54	6	6
7	(bell)	CTRL-G	55	7	7
8	(backspace)	CTRL-H or ←	56	8	8
9		CTRL-I	57	9	9
10	(linefeed)	CTRL-J	58	:	:
11		CTRL-K	59	;	;
12		CTRL-L	60	<	<
13	(carriage return)	CTRL-M	61	=	=
14		CTRL-N	62	>	>
15		CTRL-O	63	?	?
16		CTRL-P	64	@	@
17		CTRL-Q	65	A	A
18		CTRL-R	66	B	B
19		CTRL-S	67	C	C
20		CTRL-T	68	D	D
21	(forward space)	CTRL-U or →	69	E	E
22		CTRL-V	70	F	F
23		CTRL-W	71	G	G
24	(cancel line)	CTRL-X	72	H	H
25		CTRL-Y	73	I	I
26		CTRL-Z	74	J	J
27		Esc	75	K	K
28		n.a.	76	L	L
29		CTRL-SHIFT-M	77	M	M
30		CTRL- ^	78	N	N
31		n.a.	79	O	O
32	space	space bar	80	P	P
33	!	!	81	Q	Q
34	"	"	82	R	R
35	#	#	83	S	S
36	\$	\$	84	T	T
37	%	%	85	U	U
38	&	&	86	V	V
39	'	'	87	W	W
40	(	(	88	X	X
41	)	)	89	Y	Y
42	*	*	90	Z	Z
43	+	+	91	[	n.a.
44	,	,	92	\	n.a.
45	-	-	93	]	SHIFT-M
46	.	.	94	^	^
47	/	/	95		n.a.

n.a. = not available on the Apple II keyboard.

## Applesoft Reserved Word Tokens

Token	Reserved Word	Token	Reserved Word	Token	Reserved Word
128	END	164	LOMEM:	200	+
129	FOR	165	ONERR	201	-
130	NEXT	166	RESUME	202	*
131	DATA	167	RECALL	203	/
132	INPUT	168	STORE	204	^
133	DEL	169	SPEED=	205	AND
134	DIM	170	LET	206	OR
135	READ	171	GOTO	207	>
136	GR	172	RUN	208	=
137	TEXT	173	IF	209	<
138	PR#	174	RESTORE	210	SGN
139	IN#	175	&	211	INT
140	CALL	176	GOSUB	212	ABS
141	PLOT	177	RETURN	213	USR
142	HLIN	178	REM	214	FRE
143	VLIN	179	STOP	215	SCRN(
144	HGR2	180	ON	216	PDL
145	HGR	181	WAIT	217	POS
146	HCOLOR=	182	LOAD	218	SQR
147	HPlot	183	SAVE	219	RND
148	DRAW	184	DEF	220	LOG
149	XDRAW	185	POKE	221	EXP
150	HTAB	186	PRINT	222	COS
151	HOME	187	CONT	223	SIN
152	ROT=	188	LIST	224	TAN
153	SCALE=	189	CLEAR	225	ATN
154	SHLOAD	190	GET	226	PEEK
155	TRACE	191	NEW	227	LEN
156	NOTRACE	192	TAB(	228	STR\$
157	NORMAL	193	TO	229	VAL
158	INVERSE	194	FN	230	ASC
159	FLASH	195	SPC(	231	CHR\$
160	COLOR=	196	THEN	232	LEFT\$
161	POP	197	AT	233	RIGHT\$
162	VTAB	198	NOT	234	MID\$
163	HIMEM:	199	STEP		



## Conversion Tables

This appendix contains the following conversion tables:

- Hexadecimal-Binary Numbers
- Hexadecimal-Decimal Integers

### Hexadecimal-Binary Conversion Table

Use the table below to convert between hexadecimal numbers in the range 0-OF and binary numbers in the range 0000-1111.

Convert larger binary numbers to hexadecimal numbers by converting four binary digits at a time, working from right to left. If there are fewer than four binary digits in the leftmost group, add leading zeros. Here is an example:

$$100101_2 = \underbrace{0010}_{2} \underbrace{0101}_{5}_{{}_{25}_{16}}$$

Convert hexadecimal numbers larger than OF to binary one digit at a time. Here is an example:

$$\begin{array}{c} \swarrow \quad \searrow \\ 3 \quad 7 \\ \underbrace{0110} \quad \underbrace{0111} \\ \underbrace{01100111}_2 \end{array}$$

Hexadecimal	Binary
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
0A	1010
0B	1011
0C	1100
0D	1101
0E	1110
0F	1111

# HEXADECIMAL-DECIMAL INTEGER CONVERSION TABLE

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

Hexadecimal fractions may be converted to decimal fractions as follows:

- Express the hexadecimal fraction as an integer times  $16^{-n}$ , where  $n$  is the number of significant hexadecimal places to the right of the hexadecimal point.

$$0. CA9BF3_{16} = CA9BF3_{16} \times 16^{-6}$$

- Find the decimal equivalent of the hexadecimal integer

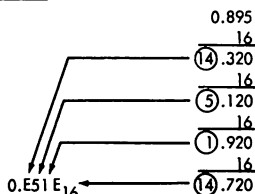
$$CA9BF3_{16} = 13\,278\,195_{10}$$

- Multiply the decimal equivalent by  $16^{-n}$

$$\begin{array}{r} 13\,278\,195 \\ \times 596\,046\,448 \times 10^{-16} \\ \hline 0.791\,442\,096_{10} \end{array}$$

Decimal fractions may be converted to hexadecimal fractions by successively multiplying the decimal fraction by  $16_{10}$ . After each multiplication, the integer portion is removed to form a hexadecimal fraction by building to the right of the hexadecimal point. However, since decimal arithmetic is used in this conversion, the integer portion of each product must be converted to hexadecimal numbers.

Example: Convert  $0.895_{10}$  to its hexadecimal equivalent



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791



## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
70	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
80	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A0	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B4	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C0	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Continued)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D0	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095



# K

## Bibliography

### **BASIC**

Albrecht, Finkle, and Brown. *BASIC*. Peoples Computer Company, Menlo Park, California, 1967.

Coan, James S. *Advanced BASIC*. Hayden Book Co., Rochelle Park, New Jersey.

Coan, James S. *Basic BASIC*. Hayden Book Company, Rochelle Park, New Jersey.

Dwyer, T. *A Guided Tour of Computer Programming in BASIC*. Houghton Mifflin Company, 1973.

Kemeny, J., and Kurtz, T. *BASIC Programming*. Peoples Computer Company, Menlo Park, California, 1967.

Pegels, C. *BASIC: A Computer Programming Language*. Holden-Day, Inc., 1973.

Peoples Computer Company. *What to Do After You Hit Return*. Menlo Park, California.

Sack, J., and Meadows, J. *Entering BASIC*. Science Research Associates, 1973.

### Assembly Language Programming

Leventhal, Lance A. *6502 Assembly Language Programming*. Osborne/McGraw-Hill, Berkeley, California, 1979.

Osborne, A. *An Introduction to Microcomputers: Volume 1 — Basic Concepts*. 2nd ed., Osborne/McGraw-Hill, Berkeley, California, 1980.

Scanlon, Leo J. *6502 Software Design*. Howard W. Sams, Indianapolis, Indiana.

Zaks, Rodnay. *6502 Applications Book*. Sybex, Berkeley, California.

### Periodicals

"Apple-Cart," *Creative Computing*. P.O. Box 789-M, Morristown, New Jersey 07960.

*Micro*, P.O. Box 6502, Chelmsford, Massachusetts 01824.

*Nibble*. P.O. Box 325, Lincoln, Massachusetts 01773.

*Purser's Magazine*. P.O. Box 466, El Dorado, California 95623.

*Softalk*. 10432 Burbank Bl., N. Hollywood, California 91601.

### Apple Publications

Apple II/II Plus Reference Manual	A2L0001A
Parallel Printer Interface Manual	A2L0004
Apple II BASIC Programming Manual	A2L0005
Applesoft II Reference Manual	A2L0006
Communications Interface Manual	A2L0007
High-Speed Serial Interface Manual	A2L0008
Disk II Floppy Disk Manual (DOS 3.2.1)	A2L0012
Applesoft Tutorial Manual	A2L0018
Graphics Tablet Manual	A2L0033
Silentype Manual	A2L0034
The DOS Manual (DOS 3.3)	A2L0036



## **Screen Layout Forms**

Use the forms in this appendix to plan the appearance of the display screen. On the text screen form, row and column numbers start with 1, which is appropriate for text work. On the low-resolution graphics screen form, row and column numbers in it start with 0, as do low-resolution graphics commands.



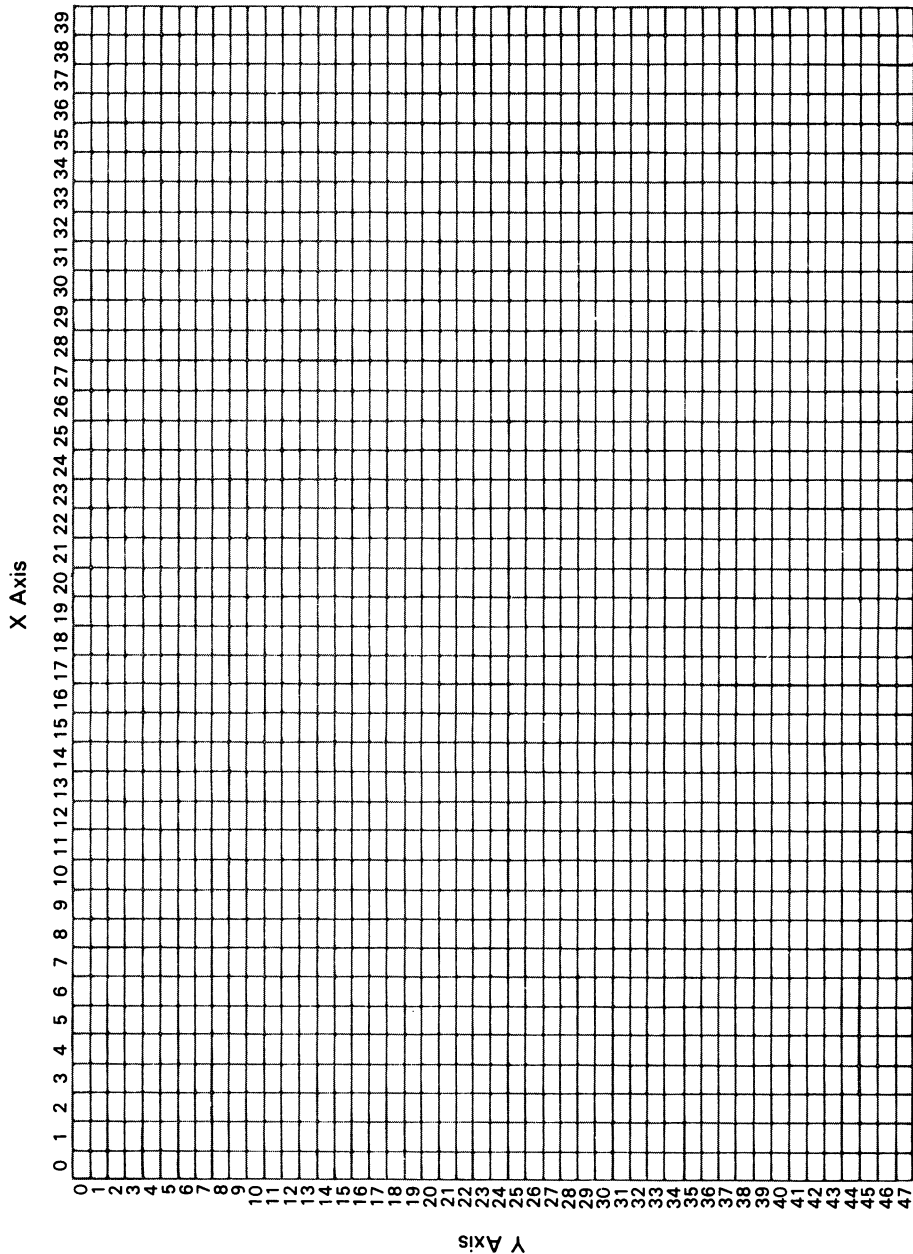


### Horizontal Tab Position

### Vertical Tab Position

## Text Screen





Low-Resolution Graphics Screen



# Index

- ABS, 312
- Accumulator, 236, 238–39
- Addressing modes, assembly language, 251
- Annunciators, game controls, 343–44
- APPEND, 186, 283. *See also* OPEN
  - machine language fix for, 264
- Applesoft. *See also* Programs
  - accessing from Monitor, 233
  - commas in, 114–15
  - line numbers, 57
  - memory usage, 352–55
  - numeric values, 40
  - program line length, 42
  - program listing, 46
  - reserved word tokens, 347, 361
  - restarting, 33–35
  - restarting via CTRL-Y, 249
  - switching to Integer BASIC, 49–50, 192
  - tab stops in, 114–15
  - variable names, 64–65
- Applesoft Firmware card, 7
- Arithmetic expressions, 70
- Arrays
  - dimensions, 67
  - index, 66
  - initialization, 146, 154
  - names, 66
  - redimensioning, 80
  - storing on cassette, 151–52
- Arrow keys, 18–19, 32
- ASC, 313
- ASCII
  - codes, 360
  - programming characters in, 122–23
- Assembly language
  - with BASIC, 106, 259–60
  - built-in subroutines, 258–60
  - debugging, 255–57
  - listing programs, 254–55
  - Mini-Assembler, 251–54
  - programs on disk, 186–87
  - relocating, 259–60
- Assembly Language Monitor. *See* Monitor
- Assignment statements, 76
- ATN, 313
- AUTO, 47, 156, 264
- Automatic line numbering, 47
- Autostart Monitor. *See* Monitor
- BASIC, 15. *See also* Applesoft; Integer BASIC
  - accessing, 28
  - with assembly language, 227–29
  - prompt characters, 38
  - restarting, 33–35
  - starting up, 29, 37–38
  - versions of, 28, 192
- Backspace. *See* Left-arrow key
- Binary arithmetic, 248–49
- Binary Image Files. *See* Files,
  - machine language
- Blank spaces, 58
- BLOAD, 196, 221, 242–43, 265
  - verifying, 247
- Boolean expressions, 73
- Booting DOS, 22–26, 49
  - Applesoft, 167
  - autostarting, 166

- Booting DOS (Cont.)
  - CTRL-K, CTRL-P Monitor boots, 167
  - Integer BASIC, 167
  - language System, 167
  - Monitor jump booting, 166
- Branching
  - BASIC statements, 81-84
  - Monitor command, 247
- BRUN, 196, 265
- BSAVE, 195, 220, 266-67
  - verifying, 247
- Calculator mode. *See* Immediate mode
- CALL, 106, 259, 266. *See also* USR
- Cards. *See* Circuit boards
- Carriage return, 94
  - as part of string, 123
- Cassettes
  - handling, 19
  - reading memory from, 240-41
  - reference tone, 240
  - saving memory on, 239-40
  - saving programs on, 47-49
  - with sound, 227
  - storing arrays on, 151-52
  - write-protecting, 20
- Cassette recorder, 19. *See also* Cassettes
  - adjusting playback volume, 20-21
  - hookup, 4
- CATALOG, 26, 168, 169, 266
- CHAIN, 267
- Character set, changing printed, 149-50
- Checksum, 176, 240-41
- CHR\$, 122-23, 150, 313
- Circuit boards
  - AppleSoft Firmware card, 7
  - communications interface card, 7
  - controller cards, 6, 7, 107-08
  - Integer BASIC card, 7
  - Language System card, 7, 14
  - main, 2-3
  - parallel printer interface card, 7
  - serial interface card, 7
- CLEAR, 79, 267
- CLOSE, 178-79, 183, 142, 268
- CLR, 79, 156, 268
- Color, 200
  - complement, 222
- COLOR=, 268
- Commas, 113-17, 142
  - in data files, 183-85
  - separating variables, 93
- Communications interface card, 7
- Compound program lines, 153
- Computed GOSUB, 91
- Computed GOTO, 82
- CON, 98, 156, 269
- Concatenation, 69
- Conditionals, 283
- Constants, 152-53
- CONT, 98, 269
- Control key. *See* CTRL
- COS, 314
- CTRL, 17-18
  - CTRL-B, 15, 28, 49, 233
  - CTRL-C, 30, 35, 46, 98, 192, 232, 233
  - CTRL-E, 235, 238-39
  - CTRL-K, 167
  - CTRL-P, 167, 248
    - Monitor boot, 25
  - CTRL-S, 46, 235
  - CTRL-X, 18, 32, 50
  - CTRL-Y, 249
- Cursor
  - control, 119-20
  - defined, 14
  - determining horizontal position, 118
  - determining vertical position, 119
  - moving, 52-53
  - positioning, 119-20
  - special video effects, 119-27
- DATA, 59-61, 77, 156, 270
- Data entry, 108, 123-42
  - allowing changes, 127-29
  - backspacing, 141-42
  - checking for errors, 129-31, 133, 135-36
  - correcting errors, 141
  - designing, 131-32
  - forms, 136
  - interactive, 124-27
  - masks, 132, 136, 138, 141-42
  - minimizing errors, 123
  - operator aids, 141-42
  - organization, 123
  - range checking, 128-29
  - style guidelines, 127-29
  - valid date, 131
- Data files. *See* Files
- Data window, 143-48
- Debugging programs, 154
  - assembly language, 255-57
  - DOS aids, 194-96
- Deferred mode. *See* Programmed mode
- DEF FN, 156, 270. *See also* FN
- DEL, 156, 271
- DELETE, 174-75, 180, 271
- DIM, 80, 272
- Direct mode. *See* Immediate mode
- Disassembled listings, 254-55
- Disk controller card, 7, 166
- Disk drive, 5, 157-67
  - drive number, 170-71
  - maximum number available, 171
  - slot number, 170-71
  - specifying in disk commands, 170-71
- Diskettes. *See also* Disk drives; Disks; DOS
  - handling, 21
  - hard sectored, 162
  - index hole, 162
  - inserting, 22
  - soft sectored, 162
  - System Master Diskette, 21, 23, 25, 26, 28
  - write protecting, 163
- Disk files. *See* Files
- Disk Operating System. *See* DOS
- Disks. *See also* Diskettes
  - binary image loading, 242-43
  - blank, 172-74
  - booting, 166-67
  - buffer, 165
  - catalog, 26
  - crash, 165
  - data storage process, 160
  - deleting programs from, 174-75

- directory, 165, 168
- floppy, 160
- hard, 156
- initializing, 27-28, 164, 172-74
- LOAD, 170
- mini-, 160
- RUN, 170
- shape tables, 220
- sectors, 160
- with sound, 226
- storage process, 165
- tracks, 160
- track/sector list, 164-65
- volume, 171-72
- Winchester, 158
- Disk II, 5, 21, 157
- Display screen
  - adjusting width, 52
  - avoid last column, 146
  - clearing, 119
  - color adjustment, 30
  - data window, 143-48
  - design, 142
  - determining screen color, 202
  - dimensions of, 2
  - 80-column, 8, 14
  - flashing video, 126
  - formatting, 142
  - full-screen graphics, 199
  - high-resolution graphics, 203-10
  - high-resolution shapes, 210-24
  - hookup, 2
  - low-resolution graphics, 197-203
  - program listing, 46
  - reverse video mode, 120, 248
  - soft switches, 341-42
  - speed, 120
  - television monitor, 1, 2
  - television set, 1, 2
  - text mode, 199
  - wrap-around, 39, 147
- DOS
  - booting, 22-26, 49, 166-67
  - memory requirements, 350
  - programmed mode, 107, 108, 148-49
  - versions of, 26, 164-65, 167
- DRAW, 222, 273
- Drive number, 170-71
- DSP, 155, 273
- Editing, 18, 32
  - adding lines, 51
  - changing characters, 53
  - changing lines, 51-55
  - deleting characters, 53
  - deleting lines, 50-51
  - edit mode, 52-53
  - erase to end of line, 54
  - erase to end of screen, 54
  - ESC sequence, 52-53
  - inserting characters, 54
- Edit mode, 52-53
- END, 43, 98, 156, 274
- Error message, 32, 40-41, 327-32
- Errors
  - checking, input, 129-31
  - correcting, 32
  - correcting during data entry, 1-41
- Mini-Assembler, 250
- Monitor commands, 238
- PEEK and POKE locations, 351
- ESC, 18, 53, 54, 56
- ESC-@, 32, 50
- Escape key. *See* ESC
- EXEC, 191-93, 274
- EXP, 314
- Expressions, 68
  - arithmetic, 70, 71
  - Boolean, 73
  - real, 71
  - relational, 71
- File pointer, 179-82
  - moving, 186, 190
- Files
  - buffers, 178, 193
  - closing, 165
  - deleting from disk, 174
  - disk, 164
  - END OF DATA, 183, 186
  - EXEC, 191
  - field positioning, 186, 190
  - field separation, 185
  - GET, 184
  - Integer BASIC/Applesoft differences, 183
  - locking, 168, 175
  - machine language, 195
  - names, 169, 181
  - random access, 187-91
  - renaming, 175
  - sectors used, 169
  - sequential, 177-83
  - storing numbers in, 185
  - types, 168
  - unlocking, 175
  - verifying, 176
- FN, 314. *See also* DEF FN
- FOR, 84, 85, 120, 156, 180, 275
- Formatting, 116
- FP, 28, 40, 276
- FRE, 154, 314
- Functions, 99-104
  - nested, 104
  - numeric, 100-01
  - string, 102
  - system, 103
  - user-defined, 103-04
- Game controls, 9, 210, 354
  - operating, 105
  - PEEK and POKE locations, 343-44
  - sensing, 105
- Garbage collection, 354
- GET, 97, 133, 156, 277
  - read text files, 184
- Ghost vectors, 211
- GOSUB, 89, 156, 277
- GOTO, 81-84, 278
- GR, 198, 279
- Graphics
  - low-resolution, 197-203
  - full screen, 199
  - high-resolution, 203-24
  - text window, 199
- Graphics tablet, 10
- Greeting program, 27, 172, 176

- Hard disk crash, 165
- Hard disks, 157
- HCOLOR, 207, 279
- Hexadecimal number
  - arithmetic, 248–49
  - conversion to decimal, 365–70
- HGR, 204–05, 280
  - alternatives to, 205–06
- HGR2, 205, 280
  - alternatives to, 205–06
- High-resolution graphics
  - clearing high-resolution pages, 206–07
  - colors, 207
  - disk files with, 195–96
  - drawing lines, 208–09
  - Integer BASIC, 205–10
  - memory requirements, 203–04
  - pages, 203–04
  - plotting points, 208–09
  - reserving memory, 203–04
  - restoring text mode, 206
  - screen dimensions, 203
  - setting up display, 204–07
  - shapes, 210–24
  - text window, 205
- HIMEM:, 106, 156, 204, 218, 221, 260, 281, 350, 352
- HLIN, 201, 282
- HOME, 282
- HPLOT, 211, 282
  - alternatives to, 209–10
- HTAB, 119, 142, 283
- IF-THEN, 91, 283
- Immediate mode, 38, 156, 261
  - arithmetic, 39
- INO, 167, 285
  - programmed mode, 107–08
- Index Register X, 236, 238–39
- Index Register Y, 236, 238–39
- Index variable, 84
- Indirect mode. *See* Programmed mode
- INIT, 172–74, 285
- INPUT, 94, 125–26, 132, 153, 156, 286
  - checking for errors, 129–31
  - disk files, 182
  - EXEC files, 191
  - mixed-type, 96
- INT, 29, 49, 287, 315
- Integer BASIC. *See also* BASIC; Programs
  - accessing from Monitor, 232–33
  - automatic line numbering, 47
  - line numbers, 57
  - memory usage, 350–52
  - numeric values, 39
  - restarting, 33–35
  - switching to AppleSoft, 49–50
  - tab stops in, 113
  - variable names, 63
- Integer BASIC card, 7
- Integer expressions, 70
- Integer variables, 64
- Interface cards, 107–08
- Intrinsic subroutines, 333–37
- INVERSE, 120, 180, 287
- Job queuing, 191
- Keyboard, 1, 16
  - deactivating via Monitor, 248
- Language System, 23
  - booting DOS, 25–26, 167
  - card, 7, 14, 28
- LEFT\$, 315
- Left-arrow key, 18–19, 32, 56
- LEN, 315
- LET, 76, 288
- Line length, 149–50
  - adjusting printed, 149–50
- Line numbers, 44, 57, 153
  - as addresses, 58
- LIST, 45, 59, 288
- Listing. *See* Program listing
- LOAD, 29, 156, 289
- Loading programs. *See* Program loading
- LOCK, 175, 290
- Location counter, 251, 252–53, 254–55, 258
- LOG, 316
- Logical operators, 73
- LOMEM:, 106, 156, 204, 260, 290, 351–52
- Loops, 84–87, 275, 292
  - nested, 85–87
- Lower-case letters, 16, 17
- Low-resolution graphics
  - colors, 197–98, 200
  - determining screen color, 202
  - full-screen graphics, 199
  - horizontal lines, 201
  - pages, 197–99
  - plotting points, 200
  - screen dimensions, 197
  - text window, 199
  - vertical lines, 201
- Machine Language Monitor. *See* Monitor
- Machine language programs, 250–60. *See also*
  - Assembly language
- MAN, 47, 156, 291
- MAXFILES, 193–94, 291
- Memory
  - address pointer, 234
  - addressing, 105
  - allocation, 260
  - altering blocks, 237
  - altering single addresses, 105, 236
  - block moves, 196, 244
  - capacity, 4
  - comparing blocks, 245
  - direct access via BASIC, 105
  - examining blocks, 234
  - examining single addresses, 105, 233
  - examining words, 234
  - filling with a pattern, 242–43, 244
  - maps, 350–54
  - random-access (RAM), 4
  - read-only (ROM), 4
  - read/write (RAM), 4
  - retrieving via Monitor, 240–41
  - saving via BASIC, 195–96, 242
  - saving via Monitor, 239
  - setting boundaries, 106, 107
  - specifying in BASIC, 105–06
  - verifying saved, 246
- MID\$, 316
- Mini-Assembler, 249–54



- accessing, 250
- exiting, 251
- instruction formats, 251–52
- location counter, 251–55
- Monitor commands, 250
- sample session, 253–54
- Mistakes. *See* Errors
- Mixed-type expressions, 74
- MOD, 70
- MON, 194, 292
- Monitor, 15, 231–51, 254–59
  - accessing, 231
  - address pointer, 235–37, 247
  - alter registers, 238–39
  - Autostart Monitor, 23, 232
  - built-in subroutines, 258–60, 333
  - commands, 238
  - defined, 6
  - examine memory, 233, 236
  - examine registers, 235
  - exiting, 232
  - fill memory procedure, 244
  - GO command, 233, 247
  - input device selection, 248
  - list command, 254–55
  - location counter, 254–55
  - move memory, 243
  - output device selection, 248
  - read memory, 240–41
  - reverse video, 248
  - saving memory, 239
  - with sound, 225
  - standard Monitor, 231
  - step command, 254–56, 258
  - trace command, 256
  - user-definable command, 249
  - verify memory, 244–46
  - verify saved memory procedure, 246–47
- NEW, 43, 156, 292
- NEXT, 84, 156, 292
- NO DSP, 155, 293
- NOMON, 194–95, 293
- NORMAL, 120, 294
- NO TRACE, 154, 294
- Null DOS command, 182
- Null string, 59
- Numbers, 60–62
  - integers, 60
  - real numbers, 60–61
  - scientific notation, 61
- Numeric values, printing as strings, 146
- Numeric variables, 64
- ONERR GOTO, 156, 183, 294
  - error handling routine, 130
  - machine language fix for, 295
  - negating, 130
- ON-GOSUB, 91, 295
- ON-GOTO, 83, 296
- OPEN 178, 187, 296. *See also* APPEND
- Operators
  - arithmetic, 68
  - Boolean, 68
  - precedence of, 68
  - relational, 68
  - string, 68
- Output
  - aligning numeric values, 146
  - array, 143
  - formatting, 142
- Paddles. *See* Game controls
- Parallel printer interface card, 7, 148
- Parentheses, 69
- PDL, 316
- PEEK, 106, 317
  - determining cursor column, 118
  - determining cursor row, 119
  - useful locations, 339
- Peripheral devices
  - input from, 107–08
  - output from, 107–08
- PLOT, 200, 297
  - in text mode, 297
- Plotting vectors, 211
  - codes, 212
  - assembling vectors by computer, 216
- POKE, 106, 260, 298
  - with graphics, 199, 205–06
  - with shapes, 219, 221
  - text window, 121–22
  - useful locations, 339
- POP, 90, 135, 298
- POS, 118, 317
- Power-on, 13
- PR#, 148, 150–51, 167, 299
  - programmed mode, 107–08
- Precedence, 68–69
  - overriding, 69
- PRINT statement, 93, 108, 154, 299
  - abbreviated, 40
  - commas in, 113–15, 142
  - disk files 179–82, 185
  - DOS commands, 107–08
  - formatting, 116–18
  - numeric data, 111–15
  - numerics as strings, 146
  - semicolons in, 109–13
  - SPC in, 117
  - strings, 109–10, 114–15
  - TAB in, 117
- Printer
  - activating via Monitor, 248
  - CHR\$ with, 150
  - control codes, 150
  - character set, 149–50
  - formatting, 142
  - hookup, 10
  - line length, 149–50
  - page size, 149–50
  - programmable, 149–50
  - selecting for output, 148
- Printing a heading, 116–17
- Printing program listings, 150–51
- Program Counter, 235, 238–39
- Program loading
  - cassette, 29
  - disk, 29–30
- Program examples
  - assembly language sound, 253
  - blanket, 127
  - display data entry form, 136–37
  - display data window, 145–46
  - end of file marker, 264

- Program examples (Cont.)
  - greeting program, 172
  - high-resolution Integer BASIC sketch, 210
  - low-resolution diagonal line, 200
  - random low-resolution lines, 202
  - shape definition, 216–18
  - sound generator, 228–29
  - speaker driver, 227
  - text window input, 122
  - valid date entry, 133–35
- Programmed mode, 38, 43, 261
  - DOS commands, 107–08, 148–49, 176–77
  - restrictions, 156
- Programming languages, 56–57
  - dialects, 57
  - syntax, 57
- Program output, 108–23, 142–51
- Programs
  - adding lines, 45, 51
  - blank spaces in, 58–59
  - changing lines, 51–55
  - compact, 153
  - debugging, 154–55
  - deleting from disk, 174–75
  - deleting lines, 50–51
  - ending, 43
  - execution, 43
  - faster, 152–53
  - line length, 39, 45
  - line numbers, 44
  - locking, 175
  - multiple statement line, 45, 153
  - optimization, 152–54
  - renaming, 175
  - saving on cassette, 47–49
  - sequence of lines, 44–45, 58
  - statements, 41
  - unlocking, 175
  - verifying, 176
- Programs, types of
  - application, 5
  - greeting, 27, 172
  - intepreter, 5
  - Monitor, 6
  - operating system, 6
- Prompt character, 15
  - Applesoft, 16
  - Integer BASIC, 15
  - Monitor, 15
- Prompt message, 96–97, 127–29
- Prompts, INPUT statement, 96
- Quotation marks, as part of string, 123
- RAM. *See* Memory
- Random-access files
  - closing, 187
  - field positioning, 190
  - opening, 187
  - read, 187
  - record length, 187
  - record number, 187
  - write, 187
- Random-access memory. *See* Memory
- Random numbers, 317
- READ, 77, 182–84, 186–87, 190–91
  - assignment statement, 301
  - disk statement, 300
  - EXEC files, 191
- Read-only memory. *See* Memory
- Read/write memory. *See* Memory
- Real variables, 65
- RECALL, 151–52
- Recopy. *See* Right-arrow key
- Recursion, 90
- Registers, 259
  - altering, 238–39
  - examining, 235
- Relational expressions, 71
- Relational operators, 71
- REM, 76, 153, 302
- RENAME, 175–76, 303
- REPT, 19, 32, 50, 56
- Reserved words, 65–66
  - tokens, 361
- RESET, 16, 17, 35, 98, 256
  - accidental, 17, 33
  - recovering from, 33
- RESTORE, 79, 303
- RESUME, 130, 156, 303
- RETURN, 156, 304
- Reverse video, 120, 248
- RF modulator, 2, 13
- RIGHT\$, 317
- Right-arrow key, 18–19, 32, 52, 54–56
- RND, 317
- ROM. *See* Memory
- ROT, 233, 304
- Roundoff, 62
- RUN
  - disk statement, 304
  - general statement, 30, 125, 156, 305
- SAVE, 156, 174, 305
- SCALE, 222–23, 306
- Scientific notation, 61
- Scratch variables, 153
- SCRN, 202, 318
- Sectors, 160
  - hard, 162
  - locating, 162
- Semicolon, 109–13, 142
  - in data files, 185
  - separating variables, 94
- Serial interface card, 7, 148
- Sequential files
  - APPEND, 186
  - closing, 178
  - opening, 178–79
  - writing to, 179–84
- SGN, 319
- Shapes
  - assembling shape table, 211
  - assembling vectors by computer, 216
  - drawing, 222
  - end of shape definition, 213
  - erasing, 222
  - loading shape table, 221
  - reserving memory for, 218
  - rotating, 223
  - saving shape table, 220
  - size changes, 222
- Shape tables, 211
  - byte coding, 212–14
  - directory, 214
  - end of shape definition, 213

- entering, 218–20
- loading, 221
- pointer to, 221
- saving, 220
- size, 214
- zero plotting vector, 213
- SHIFT, 17
- SHLOAD, 220–21, 306
- SIN, 319
- Slot numbers, 3, 6, 7, 148, 170–71
  - selecting 107–08
  - specifying in disk commands, 170
- Soft switches, 205, 341–42
- Soft disk crash, 165
- Sound, 224–29
  - with machine language, 225, 227
- SPC, 117, 319
- Speaker, 105, 224
  - PEEK and POKE locations, 341
- Special video effects, 119–22
- SPEED, 120, 307
- SQR, 319
- Stack Pointer Register, 236, 238–39
- STOP, 99, 307
- STORE, 151–52, 307
- STR\$, 320
- Strings
  - comparisons, 72
  - concatenation, 69
  - null string, 59
  - variables, 63, 64
- Subroutine examples
  - ask yes or no question, 128
  - clearing high-resolution graphics pages, 206
  - data entry mask, 132
  - display row and column headings, 146
  - enter string data, 137–38
  - error handling, 131
  - get numeric input, 128
  - get two character entry, 133
  - high-resolution Integer BASIC plot, 209
  - set high-resolution background color, 208
- Subroutines
  - built-in, 333–37
  - calling BASIC, 277, 295
  - calling assembly language, 266, 321
  - data entry, 127–29
  - exit with GOTO, 135
  - nested, 90
- System Master Diskette, 21, 23, 25–26, 28, 163, 167
- TAB
  - PRINT function, 117, 320
  - statement, 119, 142, 308
- Tab stop, 113–14
- TAN, 320
- Tape recorder. *See* Cassette recorder
- Testing programs. *See* Debugging programs
- TEXT, 308
- Text mode, plotting in, 297
- TEXT window, 121–22
  - PEEK and POKE locations, 340–41
- Tokens, 361
- TRACE, 154, 308
  - with DOS, 195
- Tracks, 160
  - locating, 162
- Truncation, 75
- Truth table, 73
- TV screen, 1. *See also* Display screen
- UNLOCK, 175, 309
- Upper-case letters, 16–17
- User-defined functions, 103, 270
- USR, 260, 321
- VAL, 321
- Variables, 152–53
  - assigning values, 124–26
  - format in memory, 351
  - integer, 64
  - names, 62, 66
  - numeric, 64
  - real, 65
  - scratch, 153
  - string, 63–64
- Vectors
  - plotting, 211–14
  - ghost, 211
- VERIFY, 176, 309
- Viewfinder technique, 144
- VLIN, 310
- Volume number, 171–73
- VTAB, 119, 142, 310
- WAIT, 310
- Wrap-around, 39, 147
- WRITE, 179–82, 186–87, 190–91, 311
- Write enable notch, 163
- Write protect notch, 163
- XDRAW, 222, 312



## **Other OSBORNE/McGraw-Hill Publications**

An Introduction to Microcomputers: Volume 0 — The Beginner's Book  
An Introduction to Microcomputers: Volume 1 — Basic Concepts, second edition  
An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors  
An Introduction to Microcomputers: Volume 3 — Some Real Support Devices  
Osborne 4 & 8-Bit Microprocessor Handbook  
Osborne 16-Bit Microprocessor Handbook  
8089 I/O Processor Handbook  
CRT Controller Handbook  
68000 Microprocessor Handbook  
8080A/8085 Assembly Language Programming  
6800 Assembly Language Programming  
Z80 Assembly Language Programming  
6502 Assembly Language Programming  
Z8000 Assembly Language Programming  
6809 Assembly Language Programming  
Running Wild — The Next Industrial Revolution  
The 8086 Book  
PET and the IEEE 488 Bus(GPIB)  
PET/CBM Personal Computer Guide, 2nd Edition  
Business System Buyer's Guide  
CP/M® II User's Guide  
Apple II® User's Guide  
Microprocessors for Measurement & Control  
Some Common BASIC Programs  
Some Common BASIC Programs — PET/CBM Edition  
Practical BASIC Programs  
Payroll with Cost Accounting  
Accounts Payable and Accounts Receivable  
General Ledger  
8080 Programming for Logic Design  
6800 Programming for Logic Design  
Z80 Programming for Logic Design







The Apple II® User's Guide  
is the key to unlocking the full  
power of your Apple II  
or Apple II Plus computer.

The Apple II User's Guide:

---

Will help you program in two versions of  
BASIC using sound, color, and graphics to  
full effect.

---

Contains detailed information on the disk drive  
and the printer.

---

Includes tips on advanced programming topics.

---

Fully describes how to use the Machine  
Language Monitor.

---

Shows how to use high resolution graphics  
with Integer BASIC.

---

Provides a compendium which thoroughly  
describes every BASIC statement, command,  
and function.

---

This book will save both time and effort.  
No longer will you have to search endlessly  
for useful information. It's all here, in the  
Apple II User's Guide, thoughtfully organized  
and easy to use.

ISBN 0-931988-46-2

